VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

REALISTICKÁ ANIMACE KOUŘE

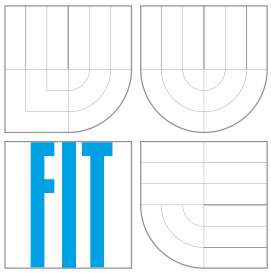
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

MILOŠ ZUBAL

BRNO 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

REALISTICKÁ ANIMACE KOUŘE

REALISTIC SMOKE ANIMATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — MILOŠ ZUBAL
AUTHOR

VEDOUCÍ PRÁCE — Ing. STANISLAV SUMEC, Ph.D.
SUPERVISOR

BRNO 2007

Abstrakt

Práce provádí základní analýzu historických a současných algoritmů pro animaci kouře. Dále jsou popsány moderní přístupy k zobrazování volumetrických dat. Na základě této analýzy jsou vybrány algoritmy použité při implementaci realistické animace kouře. Tyto algoritmy jsou podrobněji popsány a jsou zdůrazněny jejich důležité vlastnosti vzhledem k zaměření práce. Dále je podrobně popsána implementace těchto algoritmů a je provedeno měření výkonnosti. V závěru je zhodnocen dosavadní vývoj práce a jsou nastíněna další možná pokračování projektu.

Klíčová slova

kouř, dynamika tekutin, Eulerovy rovnice Navier-Strokeovy rovnice, vorticity confinement, semi-Lagrangeovské metody, volume rendering, ray tracing, texture mapping, OpenGL, LaTeX

Abstract

This work makes basic analysis of historical and current algorithms for smoke animation. Modern approaches to rendering volumetric data are briefly described. We choose algorithms for implementation on basis of this analysis. These algorithms are described in detail and we make emphasis on their important properties according to dedication of this work. Detailed description of implementation follows along with performance measurement. Conclusion evaluates results of work and proposes possible extensions.

Keywords

smoke, fluid dynamics, Euler equations, Navier-Stroke equations, vorticity confinement, semi-Lagrangian methods, volume rendering, ray tracing, texture mapping, OpenGL, LaTeX

Citace

Miloš Zubal: Realistická animace kouře, diplomová práce, Brno, FIT VUT v Brně, 2007

Realistická animace kouře

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Stanislava Sumce.

. . . . . .
Miloš Zubal
20. května 2007

Poděkování

Chtěl bych poděkovat panu Sumcovi za vynikající spolupráci při tvorbě této práce.

© Miloš Zubal, 2007.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

ZADÁNÍ DIPLOMOVÉ PRÁCE

REALISTICKÁ ANIMACE KOUŘE

Vedoucí: Sumec Stanislav, Ing., Ph.D., UPGM FIT VUT
Řešitel: Zubal Miloš, Bc.
Obor: Počítačová grafika a multimédia
Kategorie: Počítačová grafika
Zadání:

1. Prostudujte a popište existující algoritmy animovaného zobrazování kouře.
2. Prostudujte algoritmy zobrazování objemových dat vhodné pro zobrazování kouře.
3. Vyberte, analyzujte a popište algoritmus vhodný pro vytváření realistických animací kouře.
4. Implementujte vybraný algoritmus.
5. Demonstrujte funkčnost implementovaného algoritmu na několika příkladech animace kouře.
6. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu.

LICENČNÍ SMLOUVA

Licenční smlouva je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně

Obsah

Kapitola 1

Úvod

Kouř patří mezi jedny z nejzajímavějších přírodních jevů, které můžeme sledovat. Existuje v mnoha formách (oblaka, kouř z ohně, mlha atd.), které mohou lidé pozorovat v běžném životě.
Z historického hlediska je kouř fenoménem, který byl mnohdy spojován s nadpřirozenými jevy a postavami (duchové bývají někdy popisování jako "živý" kouř, transformace Džina z Aladinovy lampy do jeho lidské podoby je také prezentována pomocí kouře). V dnešní době je kouř spojován s mnoha (většinou negativními) jevy, jako je například smog nebo cigaretový kouř.
Kouř jako přírodní jev podléhá mnoha fyzikálním zákonům a lze jej považovat za velice komplexní jev.
Z hlediska počítačové grafiky je tedy potřeba chápat, že tvorba kvalitní simulace kouře je vždy balancováním na hraně mezi dostupným počítačovým výkonem a vizuální kvalitou výsledného obrazu. Kvalita animace je klíčová, protože (jak je uvedeno výše) člověk se s kouřem běžně setkává a dokáže velice citlivě reagovat i na drobné nesrovnalosti ve výsledném obrazu - nicméně úroveň kvality, potřebnou pro dobrý subjektivní vjem člověka, lze považovat za konstantní. Naproti tomu se výkon hardwaru (jak CPU, tak GPU) neustále zvyšuje a nabízí nejen vyšší rychlost fyzikální simulace (CPU), ale také rychlejší a snadnější cestu k zobrazení výsledků simulace (GPU). Tímto se nám otevírá cesta k moderním algoritmům a metodám pro jejich zlepšení.
Úkolem této diplomové práce bylo prozkoumat historické a moderní metody animace kouře a zvolit metodu vhodnou pro implementaci (samozřejmě dostatečně vizuálně kvalitní). Dalším úkolem bylo prozkoumat současné metody zobrazování volumetrických dat a zvolit metodu vhodnou pro zobrazení dat získaných předchozím algoritmem. Hlavním cílem práce byla implementace tohoto algoritmu, experimentování s ním a výsledná analýza a prezentace výsledků.
V kapitole 2 *Animace kouře* budou rozebrány historické metody pro simulaci a zobrazení kouře a bude zhodnocen jejich význam a vliv na moderní metody. Ty budou následovat a budou hodnoceny jejich důležité vlastností zvláště s ohledem na rychlost simulace a na výslednou kvalitu obrazu. Popis v této kapitole bude obecný a měl by čtenáře intuitivně seznámit s problematikou.
Úkolem kapitoly 3 *Zobrazení objemových dat* bude zhodnocení současných algoritmů pro zobrazení objemových dat (nebudeme se zabývat historickými metodami). Zde bude opět kladen důraz na porovnání z hlediska vizuální kvality a výkonu. Důležitým faktorem také bude obtížnost implementace a možnost využití schopností moderních akceleračních karet. Budou rozebrány základní pricnipy těchto metod (k detailům dospějeme až v dalších kapi-

tolách).
V kapitole 4 *Analýza* dojde k shrnutí poznátků z obou předchozích kapitol a k výběru nejvhodnější metody animace a zobrazení. Dále bude proveden výběr nejvhodnější grafické knihovny. V této kapitole se již dostaneme k matematickému popisu simulace a nástinu jeho řešení pomocí počítače (které bude podrobně rozebráno v následující kapitole).
V nejobsáhlejší kapitole 5 *Návrh a implementace* bude detailně popsán přechod od matematického modelu k simulačnímu (výpočetnímu) modelu a výsledné aplikaci. To bude zahrnovat UML diagram použitých tříd (a popis rozhraní pro použití v jiných aplikacích) a zhodnocení jednotlivých tříd a jejich úloh v simulaci. Zvláštní úsilí bude věnováno popisu procesu diskretizace a výpočtu jednotlivých fyzikálních jevů (jako například advekce). Text budou doplňovat hojně použité obrázky (z velké části vytvořené pomocí vyvinuté aplikace) a také části kódu (nebo pseudokódu).
V kapitole 6 *Výkon* bude prezentována výkonnostní analýza vzniklé aplikace. Zvláštní důraz bude kladen na analýzu trvání jednotlivých podkroků simulace. Dojde k porovnání rychlosti simulace se simulacemi v jiných článcích.
V kapitole 7 *Závěr* bude práce hodnocena jako celek a dojde k sumarizaci dosažených výsledků. Zhodnotím usílí, které bylo vyvinuto při tvorbě aplikace a provedu nástin možných rozšíření či vylepšení práce.
V rámci semestrálního projektu byly nastudovány výše uvedené metody a byla provedena jejich analýza. Na základě této analýzy byly vybrány metody a algoritmy pro simulaci a zobrazení realistických kouřových efektů. Diplomová práce na tyto výsledky navazuje implementací vybraných metod a jejich detailním popisem. Dále byla provedena analýza vyvinuté aplikace z hlediska kvality výstupu i z hlediska výkonu.

Kapitola 2

Animace kouře

V této kapitole blíže prozkoumáme historické přístupy k animaci kouře, osvětlíme jejich výhody a nevýhody a ujasníme si jejich vliv na moderní metody animace kouře. V druhé části se budeme věnovat současným, nejmodernějším, metodám simulace kouře. Popis bude spíše obecný a intuitivní - přesnějšímu popis se budu věnovat až v kapitole 4 Analýza.

2.1 Historické metody animace kouře

Z historického hlediska byla animace kouře nejpravděpodobněji použita poprvé v počítačových hrách. Tyto první pokusy byly animovány čistě rukou animátora a nebyl použit absolutně žádný fyzikální model. Z těchto důvodu můžeme tento druh animace považovat za úplně základní. Rychlost animace je v tomto případě závislá na rychlosti zobrazování jednoduchých textur (která je v dnešní době téměř zanedbatelná). Kvalita obrazu je silně závislá na zručnosti grafika/animátora, ale dosažení alespoň trochu realistického vzhledu je touto formou vpodstatě nemožné.
Na obrázku 2.1 si můžeme prohlédnout takovéto kouřové efekty z legendární hry Golden Axe.

Obrázek 2.1: Ukázky kouřových efektů ze hry Golden Axe (1989).

2.1.1 Procedurální generování kouře s využitím textur

Pod tuto sekci bych chtěl zahrnout metody, které sice používají základních principů částicových metod, ale na tak jednoduché úrovni, že jsem cítil potřebu je oddělit od ostatních. Zde je nosným prvkem animace textura kouře (vytvořená grafikem, případně generovaná), která je potom v různých variacích (roztahování, rotování) umísťována do scény tak, aby vznikl dojem kouře. Z fyzikálního hlediska se většinou předpokládá pouze fakt, že se kouř s postupem rozptyluje - textury se roztahují a zvyšuje se jejich průhlednost. Na obrázku 2.2 můžeme vidět ve středu na pravo řadu obláčků (vzniklých průletem rakety) - jedná se ve všech případech o stejnou texturu, která rotuje a s časem se zvětšuje a stáva se průhlednější až do úplného zmizení, což je klasickým případem, který reprezentuje tyto jednoduché metody.

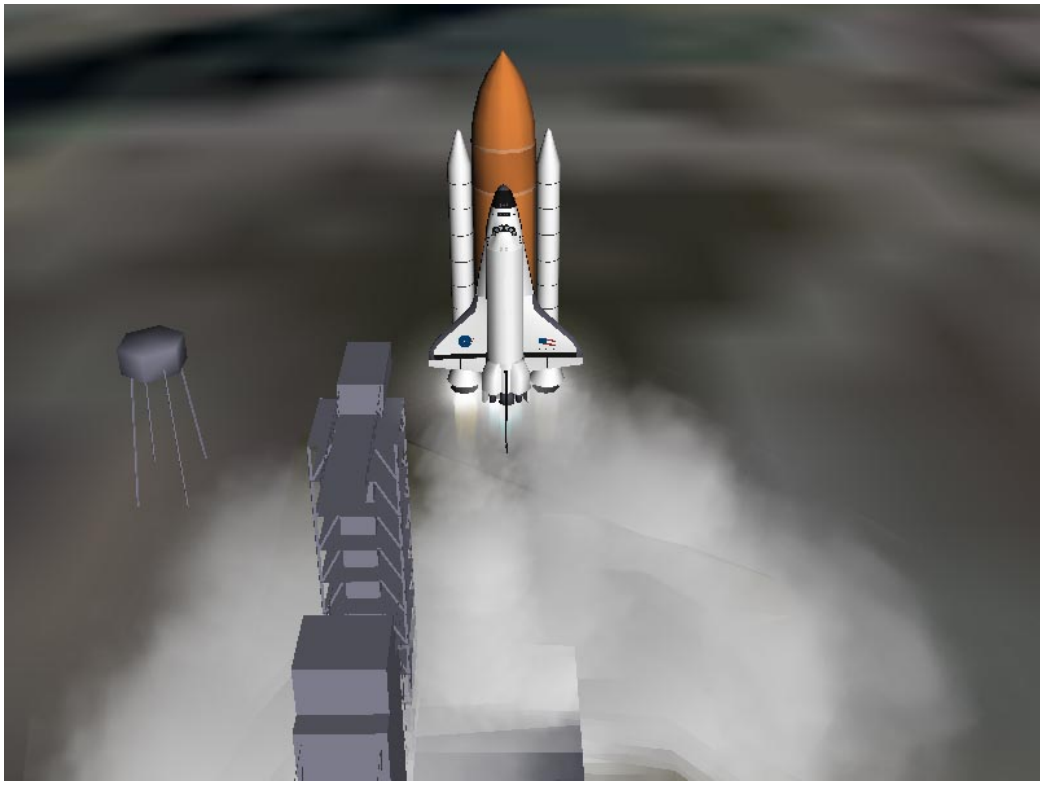
Tyto metody předpokládají renderování, které podporuje alpha blending (vykreslování

Obrázek 2.2: Ukázky kouřových efektů ze hry Quake 3 (1999).

průhledných útvarů). Zároveň je velice vhodné pokud máme jednoduché prostředky pro operace nad texturou (roztažení, translace, rotace, změna alpha kanálu). Rychlost metody tedy závisí na tom do jaké míry jsou tyto záležitosti podporovány grafickým HW (v dnešní době standart). Tento způsob animace je o dost blíže k reálnému obrazu kouře, než předchozí skupina metod. Kvalita je však dostatečná pouze v počítačových hrách, kde je rychlost rozhodující. Pro obyčejného pozorovatele není těžké na první pohled poznat, že se nejedná o reálný kouř.

2.1.2 Částicové metody

Částicové systémy nabízejí bohaté možnosti pro simulaci mnoha rozličných fyzikálních jevů mezi které patří i kouř.
Částicový systém je tvořen emitorem, který podle jeho vlastností vysílá (emituje) do prostředí autonomní částice, které potom v systému figurují. Každá částice nese informaci o svých vlastnostech. Ty se mohou lišit podle konkrétní aplikace částicového systému, ale v zásadě vždy musí být známa alespoň poloha a rychlost. Další charakteristickou vlastností částic je životnost - částice po určitém čase ze systému zmizí. V každém výpočetním kroku tedy emitor vytvoří nové částice (přiřadí jim počáteční sadu vlastností), některé částice zaniknou a pro zbytek částic se vypočítá jejich další stav. V počítačové grafice také dojde k vykreslení (které však není zcela povinné v každém kroku).
Z hlediska simulace kouře se částicím přidává řada vlastností, které odpovídají realným vlastnostem částeček kouře. Mezi tyto patří zejména barva, velikost, průhlednost, vznosná síla (dalo by se říct hustota). Částice jsou opět prezentovány texturami (ale v porovnání s předchozí metodou menšími). Systém jako takový může disponovat atmosferickými vlastnostmi, které jsou zajímavé z hlediska kouře, jako je například vítr.
Příklad částicového kouře je na obrázku 2.3. Lze pozorovat, že kvalita už je poměrně dobrá. Nicméně pořád trpí určitými nedostatky, které jsou zřetelné - kouř se jakoby "zakusuje" do objektů a působí podivným jednolitým dojmem.
Částicové metody už jsou náročnější na výpočetní výkon (jak CPU, tak grafické karty) a



Obrázek 2.3: Ukázka částicového kouře vytvořeného pomocí systému Orbiter [9].

představují současné maximum, které se dá použít při real-timovém použití animace kouře pro větší scény. Nejdůležitějším faktorem ovlivňujícím výkon, ale také kvalitu obraz, je

počet částic v systému. Kvalita obrazu je výborná (zejména statické obrazy), nicméně obsahuje některé zásadní nedostatky. Mezi ně patří zejména absence reakce na pohyb předmětů kouřem, kouř nevytváří zcela obvyklé turbulentní jevy, a proto zejména při animaci kouře po několika okamžicích člověk pozná, že se nejedná o reálný kouř.
Za výhodu oproti modernějším metodám se dá považovat rychlost a také fakt, že efekt není teoreticky prostorově omezen (tak jako tomu je u moderních metod, viz výše).

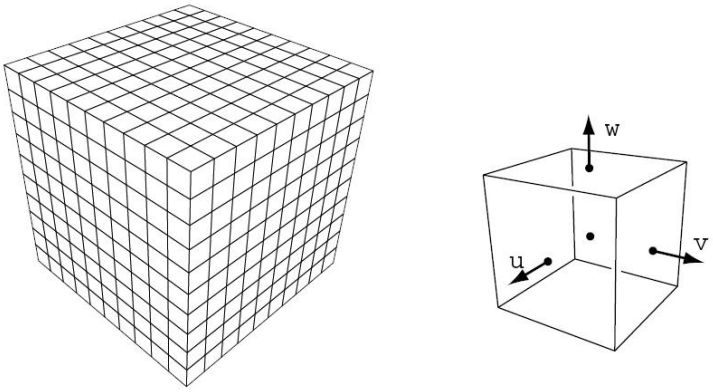
2.2 Moderní metody animace kouře

Pojmem moderní metody budu nazývat metody, které přímo simulují dynamické toky v kouři. První pokusy s touto problematikou vznikly v roce 1984 (Kajiya [8]), ale počítačový výkon v té době neumožňoval produkovat prezentovatelné výsledky. Pokračování ve vývoji této metody (co se týče hlavně zobrazování) přišlo až v roce 1996 (Foster a Metaxas [6, 7]). Mezitím došlo k vývoji v numerické simulaci toků v tekutinách. Metody počítačové grafiky ze studií toků v tekutinách čerpají, ale z důvodů rychlosti zobrazení provádí v těchto přístupech různé úpravy.
Obecně se pro tento typ simulace využívá přímo rovnic pro popis dynamických toků v tekutinách. Analytické řešení těchto složitých integrálů je nemožné a tak se používá diskretizace prostoru a numerického řešení těchto rovnic. Počítačová grafika (oproti přesné simulaci toků) používá hrubšího dělení prostoru a předpokládá jisté vlastnosti tekutiny (v závislosti na konkrétní aplikace), které pomáhají zjednodušit rovnice a tím zrychlit jejich numerické řešení.

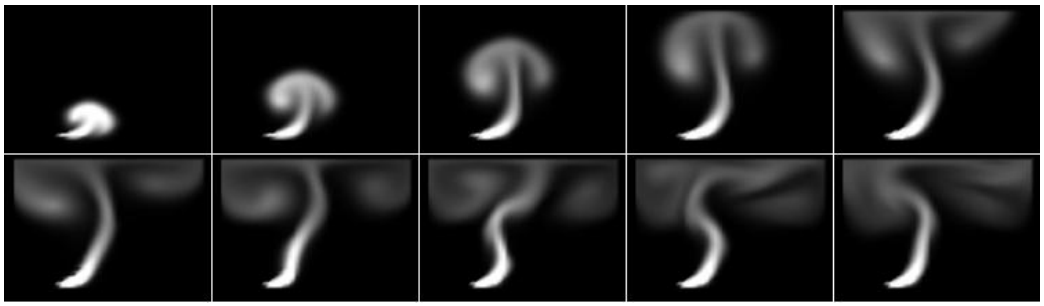
2.2.1 Fyzikální model

Fyzikální model tedy definuje prostorovou mřížku, ve které jsou v každé diskrétní jednotce definovány vlastnosti. V případě kouře to je tedy zejména hustota a rychlost. Dalším velmi vhodným parametrem je teplota, která nám umožní přímo fyzikálně simulovat s ní spojené jevy. Na obrázku 2.4 vidíme znázornění diskretizace prostoru a definování vlastností (rychlost v případě naší implementace ale není definována na stěnách voxelu, ale taktéž v jeho středu).

Změna stavu v každém výpočetním kroku tedy znamená provedení numerického vypočtení rovnic za účelem získání nového stavu vlastností v celé mřížce.
Existuje několik přístupů jak řešit tyto rovnice. V zásadě lze použít implicitní nebo explicitní řešení rovnic. Explicitní řešení je jednodušší, ale méně přesné. V úvahu je také potřeba vzít řád integrace, kterým se dá zvýšit přesnost, ale také snížit rychlost výpočtu. Další možnou alternativou k řešení jsou celulární automaty (použito například v [3]). Hlavním výstupním parametrem voxelu pro zobrazení bude tedy hustota (ale lze uvažovat i o teplotě). Samotné zobrazování takto generovaných volumetrických dat bude probráno v další kapitole. Pro ukázku výstupu jsem tentokrát zvolil 2D obraz, protože v něm lze lépe pozorovat turbulentní jevy, které se takto dají simulovat. Na sekvenci obrázků 2.5 [10] tedy vidíme, jak probíhá simulace s použitím moderních metod.



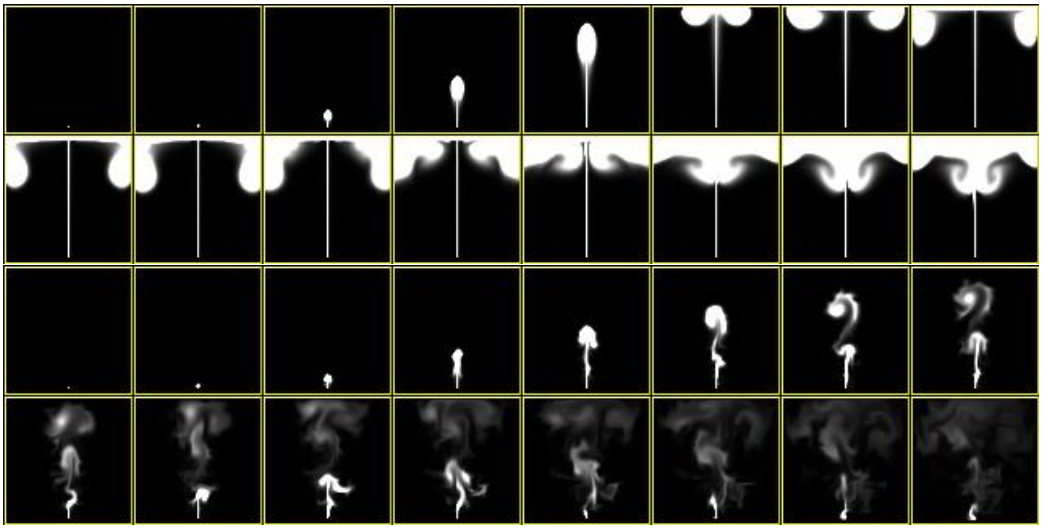
Obrázek 2.4: Ukázka diskretizace prostoru a definice vlastností jednotlivých voxelů.



Obrázek 2.5: Průběh simulace dynamických toků ve 2D.

2.2.2 Metody pro vylepšení vizuální kvality

Kvůli zrychlení výpočtů existuje v této oblasti logická tendence ke snižování počtu voxelů v mřížce. Tím ovšem simulace ztrácí na kvalitě a roste vliv numerického rozptýlení některých lokálních detailů, které jsou velice důležité pro kvalitní a realistickou animaci kouře. Mezi tyto detaily patří zejména malé (lokální) turbulence, které jsou základním kamenem toho, aby kouř skutečně vypadal jako "živý".
Řešení se nabízí hned několik. Asi nejjednodušším je zjemnit mřížku tak, že by se neprojevily negativní vlivy numerického rozptýlení. Toto řešení je v rozporu s našimi rychlostními požadavky, a proto se jím nebudeme zabývat. Problém se ztrátou lokálních turbulencí se však dá vyrešit i na relativně hrubých mřížkách. Hlavní ideou je vracet energii ztracenou rozptýlením zpátky do modelu. Jednou cestou je náhodné generování těchto turbulencí, což však může mít za následek vizuálně nekorektně vypadající turbulence. Kvůli tomu byl vyvinut algoritmus *vorticity confinement*, který je popsán v [13]. Algoritmus dokáže rozpoznat místa, kde se vlivem rozptýlení ztratily důležité lokální detaily.
Na obrázku 2.6 můžeme pozorovat velice výrazný rozdíl v kvalitě výsledného obrazu, který je způsoben využitím metody *Vorticity Confinement*.



Obrázek 2.6: Ukázka rozdílu při použití *Vorticity confinement* [10].

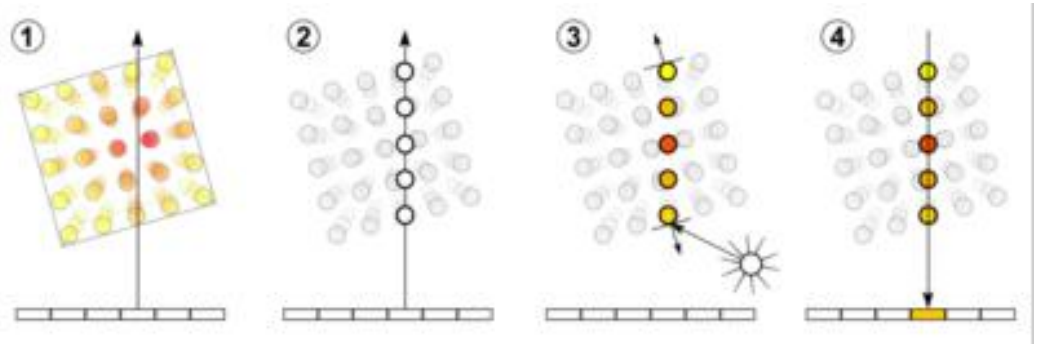
Kapitola 3

Zobrazení objemových dat

Zobrazování volumetrických dat je z historického hlediska velice náročná (jak výpočetně, tak implementačně) technika. Hlavním smyslem je projekce trojrozměrných dat na dvojrozměrnou vykreslovací plochu. Náročnost spočívá především ve velkém objemu dat a v ne zcela triviální transformaci objemových dat na dvojrozměrná výstupní data.
Existuje mnoho různých přístupů k vykreslování objemových dat, které se liší náročností implementace, výkonem a vizuální kvalitou. Bylo vyvinuto také mnoho metod pro optimalizaci vykreslování objemových dat, ale vzhledem k faktu, že je dnes lehce dostupná hardwarová podpora a že volume rendering není hlavním bodem této práce, se těmito technikami nebudu zabývat.

3.1 Ray-casting

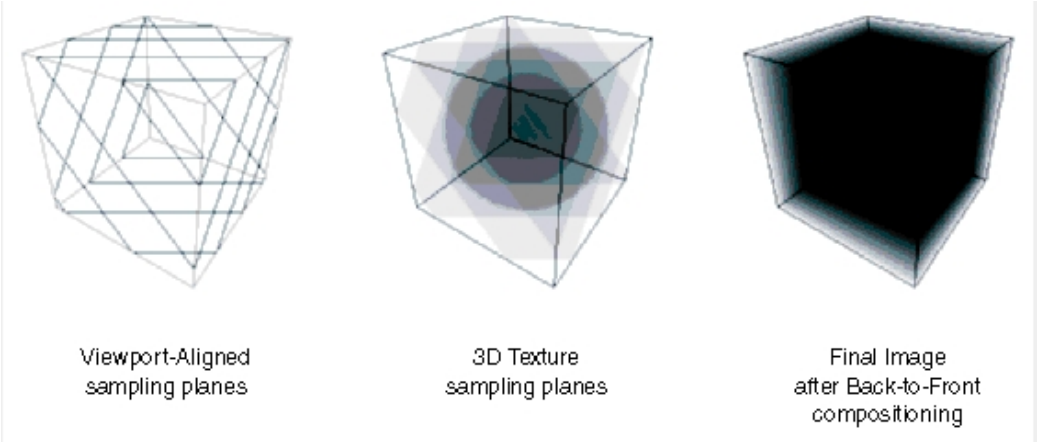
Ray-casting je intuitivní metodou jak zobrazovat volumetrická data. Všechny ostatní metody fungují vpodstatě na stejném principu, ale výsledku dosahují jinou cestou.
Tak jako v běžném ray-castingu se i zde promítají do scény paprsky, které vycházejí z kamery, procházejí promítací rovinou a protínají zobrazovaný prostor. Na rozdíl od obvyklého ray-castingu se ovšem pravidelně vzorkuje (tri-lineární interpolací) procházený objem, u každého takové vzorku se spočítá normála, provede se stínování vzorků (shading) a nakonec se postupně odzadu vzorky míchají (blending) za pomocí hodnoty alpha až vznikne barva jednoho pixelu na promítací rovině. Všechny tyto kroky jsou zobrazeny na obrázku 3.1 [1].
Toto zpracování je velice náročné na výkon (zejména interpolace a shading), a proto je velmi vhodné pokud jsou alespoň některé operace podporovány hardwarově.



Obrázek 3.1: Postupné kroky *Volume Ray castingu*.

3.2 Texture mapping

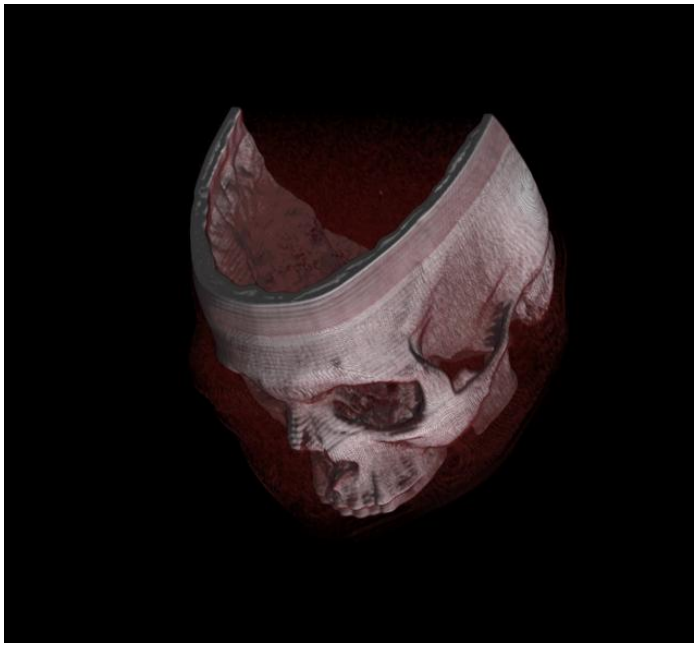
Texture mapping je metoda, která využívá schopností nových grafických karet. Moderní grafické karty podporují 3D textury. Důležitou schopností (z hlediska vykreslování volumetrických dat) je provádění řezů skrze 3D texturu a automatická (a hlavně HW podporovaná tri-lineární interpolace takto vzniklých 2D textur). Hlavní ideou je "rozřezání" 3D textury na plátky, které svou normálou směřují ke kameře a postupné vykreslení těchto plátku odzadu za pomocí alpha blendingu. Oproti ray-castingu tedy odpada výpočet normály a shading (což se projeví horší kvalitou), ale za to je tahle metoda rychlejší a mnohem jednodušší na implementaci. Tato metoda trpí artefakty způsobeným "nakrájením" textury, ale tyto nepůsobí příliš rušivě.
Na obrázku 3.2 je znázorněn základní princip Texture mappingu a na obrázku 3.3 lze vidět výsledky této metody (i se zmiňovanými artefakty) [2].



Obrázek 3.2: Demonstrace principu Texture mappingu.

3.3 Fotonové mapa

V [4] je použit speciální renderer, který využívá techniky fotonové mapy.
Fotonová mapa je technika vyvinutá Henrikem Wann Jensenem (spoluautor [4]), která se používá pro zobrazení průhledných objektů, jako je například voda, sklo, pára (a samozřejmě také kouř).
Hlavní myšlenkou metody je vyslání určitého množství fotonů (ze světelného zdroje) do scény. Pro všechny tyto fotony se počítá jejich interakce s objekty (odraz, lom, pohlcení apod.) a každá tato interakce se uloží do struktury, která se nazývá fotonová mapa. V této struktuře je uloženo místo dopadu fotonu, jeho původní směr a energie.
Každý foton může po některých interakcích dále pokračovat scénou (se zmenšenou energií a změněným směrem) a přispívat tak dalšími záznamy do fotonové mapy. Tímto způsobem vzniklá struktura je potom později použita pro výpočet odlesků ve scéně.
V našem případě je potom na kouři patrné odkud na něj dopadá světlo. Je možné takto také kontrolovat míru průchodu světla kouřem — nastavením fyzikální veličiny zvané *albedo* (v našem případě představuje míru pohlcení fotonu částečkou kouř). Můžeme tedy modelovat vizuální stránku kouře jako například tmavý, neprusvitný kouř (z prachových částic, které příliš neodráží světlo) nebo naopak světlý, průsvitný kouř (z částic vody, které světlo téměř



Obrázek 3.3: Ukázka vykreslení obejmových dat pomocí Texture mappingu.

úplně odrážejí).
Ukázku kouře renderovaného pomocí fotonové mapy můžeme vidět na obrázku 3.4.

Obrázek 3.4: Kouř renderovaný s pomocí metody fotonové mapy. (převzato z [4])

Kapitola 4

Analýza

V této kapitole budou na základě shromážděných informací vybrány animační a zobrazovací algoritmy, které budou použity při implementaci realistické animace kouře. Dále bude proveden výběr nástrojů a prostředků za jejichž pomocí bude implementován program. U všech rozhodnutí bude vyvinuta snaha zdůvodnit volbu a nastínit hlavní výhody.

4.1 Výběr animačního algoritmu

Z hlediska zadání a požadované kvality obrazu je zcela jasné, že bude použito moderního přístupu, který přímo modeluje toky v tekutině.
Pro potřeby simulace kouře bude předpokládáno, že simulovaný kouř se chová jako neviskózní a nestlačitelná (je čerpáno z [4]) tekutina. Viskozita je pro naše potřeby zcela zanedbatelným jevem (a navíc absolutně ztrácí smysl při hrubších diskretizacích prostoru). Stlačitelnost lze zanedbat pro rychlosti pohybu kouře menší než je rychlost zvuku (což pro naše potřeby stačí - jiná situace by nastala u simulací výbuchů, kde se projevují akustickotlakové vlny).

4.1.1 Rovnice proudění v tekutině

Za daných předpokladů již můžeme přejít ke konkrétním fyzikálním rovnicím, které modelují proudění v tekutině.
Pro naše potřeby se tedy jako nejlepší jeví Eulerovy rovnice (které jsou speciálním případem Navier-Strokeových rovnic - neobsahují popis viskozity) pro nestlačitelnou tekutinu [5] [4]:

$$\nabla \cdot \mathbf{u} = 0 \tag{4.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \nabla p + \mathbf{f} \tag{4.2}$$

Rovnice popisují fakt, že rychlost v tekutině by měla zachovávat hmotu (rovnice 4.1) a hybnost (rovnice 4.2). Dále je třeba popsat chování teploty a hustoty kouře. Pro naše účely bohatě stačí předpoklad, že tyto dvě veličiny se pohybují podle směru pohybu (rychlosti) rychlosti a tudíž

$$\frac{\partial T}{\partial t} = -(\mathbf{u} \cdot \nabla)T \tag{4.3}$$

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho \tag{4.4}$$

Další důležitá rovnice popisuje vliv hustoty a teploty kouře na vznosnou sílu. Ta je použita k modelování jevů, jakými je těžký, hustý, klesající kouř nebo horký, rychle stoupající, kouř.

$$f_{vznos} = -\alpha\rho\mathbf{z} + \beta(T - T_{prost})\mathbf{z} \tag{4.5}$$

parametry α a β jsou konstanty, které by měly být nastaveny tak, aby rovnice měla fyzikální smysl.

4.1.2 Numerické řešení rovnic a diskretizace

Numerické řešení spočívá v diskretizaci prostoru na konečnou mřížku voxelů v jejichž středech je definována teplota, hustota a vnější síly. Během implementace jsem dospěl k závěru, že bude jednodušší ve středu voxelu ponechat i rychlostní vektory. Oddělení rychlosti do stěn voxelů by přineslo větší odolnost numerického řešení vůči chybám a artefaktům, ale jednoduchost a rychlost je z našeho hlediska důležitější (na obrázku 2.4 jsou znázorněny rychlosti ve stěnách).
Vhodným řešením je udržovat tyto mřížky dvě. V tomto případě se během výpočetního kroku vypočte další mřížka z té aktuální a obě mřížky se prohodí (podobný princip jako double buffering v počítačové grafice).
V každém výpočetním kroku se nejdříve vypočtou nové rychlosti. Tato ne zcela triviální záležitost probíhá obvykle ve třech krocích. V prvním kroku se k novým rychlostem přičtou externí síly (např. uživatelem definované), vznosná síla (rovnice 4.5) a další síly (v našem případě síly dané algoritmem *vorticity confinement*). Toto se děje jednoduše tak, že se tyto síly vynásobí časovým krokem a přičtou se k rychlosti. Dalším krokem je advekce stávajících rychlostních vektorů. Pomocí semi-Lagrangeovského schématu a předchozího stavu vypočteme semi-Langrangeovskou cestu, z které zjistíme jakou hodnotu by měl rychlostní vektor v minulém stavu, ale na aktuální pozici. Tato hodnota se pak lineární interpolací převede zpět do středu voxelu. Nakonec je potřeba, aby rychlostní vektory splňovaly podmínku zachování hmoty. Toho dosáhneme řešením Poissonovy rovnice tlaku.
Po provedení advekce rychlostních vektorů se provede advekce teploty a hustoty podle těchto vektorů. To se děje podobným způsobem jako advekce rychlostních vektorů.

4.1.3 Vorticity confinement

Tato metoda je již zmíněna výše (metoda je popsána Steinhoffem [13]). Metoda definuje vír v proudění tekutiny a vektory, které ukazují směr z menší koncentrace víření do vyšší. Z těchto dvou veličin se pak následně počítá síla (která se z hlediska výpočtu vnímá jako externí) díky níž se do modelu vrací energie ztracená diskretizací. Tato síla pak *oživuje* kouř a dodává energii pro vznik nových lokálních víření tam, kde došlo k jejich ztrátě.

4.1.4 Zhodnocení metody

Výběr metody byl proveden hlavně na základě práce Fedkiwa [4]. Mezi hlavní výhody patří jednoduchost a rychlost. Metoda se zaměřuje zejména na rychlost výpočtu a následné *umělé* vylepšení obrazu. Z tohoto hlediska není příliš vhodná pro přesnou simulaci toků v kapalině, ale je naopak velmi vhodnou pro počítačovou grafiku (kde nároky na úplnou realističnost

nejsou). Na obrázku 4.1 si můžeme prohlédnout alespoň statické vlastnosti této metody (animace je samozřejmě mnohem více vypovídající a ještě více zdůrazňuje kvalitu této metody).

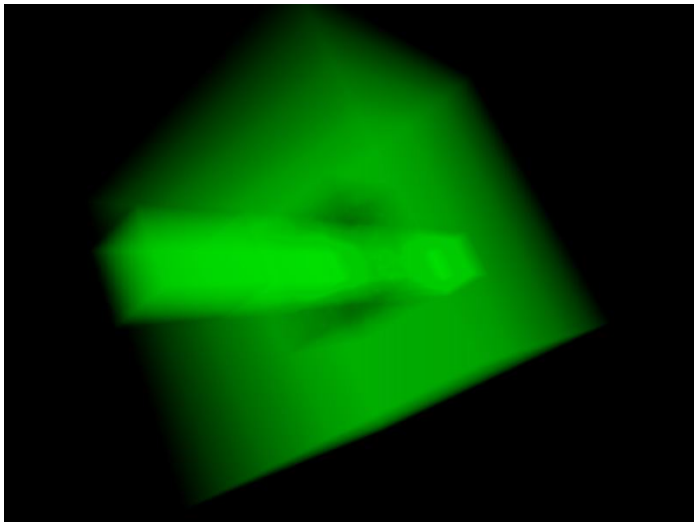
Obrázek 4.1: Ukázka vygenerovaného realistického kouře (ze článku [4]).

4.2 Výběr zobrazovacího algoritmu

Jako zobrazovací algoritmus byl vybrán *texture mapping*. Základní a postačující popis je již proveden v kapitole 3 Zobrazení. Tato metoda byla vybrána proto, že je velice jednoduchá na implementaci a díky přímé hardwarové podpoře je i poměrně rychlá. Tuto metodu se již podařilo implementovat v C++ a OpenGL a ukázku grafického výstupu si můžete prohlédnout na obrázku 4.2.
Metoda má nedostatky v kvalitě grafického výstupu. Vytváří artefakty, což však není výrazným problémem (zvláště u zobrazování kouře). Závažnějším problémem je absence práce s vrženým světlem a jeho odrazem od částic kouře, protože tento efekt je velmi důležitou součástí vyvolání iluze skutečného kouře.
Nakonec bylo učiněno rozhodnutí, že dokonalejší renderer bude vyvíjen pouze při dostatečné časové rezervě (např. v případě, že se povede rychlá a úspěšná implementace animačního algoritmu). Následně se během implementace ukázalo, že pro demonstraci schopností simulační metody bude postačovat *Texture Mapping*. *Volume Ray-Tracing* je dle mého názoru nezbytný pro generování realisticky vypadajících animací, ale jeho implementace je mimo rozsah této práce.

4.3 Výběr grafické knihovny a jazyka

Výběr grafické knihovny a jazyka je nejjednodušším krokem. V oblasti počítačové dlouhodobě platí za standardy OpenGL a C++ a použité prostředky se v tomto ohledu nebudou odlišo-



Obrázek 4.2: Ukázka výstupu z již implementovaného volume rendereru (založen na texture mappingu).

vat.
C++ je vybráno zejména z důvodu jeho velice široké podpory různými knihovnami, rychlosti a v neposlední řadě také podpoře objektového přístupu (který jsem při implementaci s výhodou použil).
OpenGL je taktéž široce podporováno knihovnami (zejména nadstavbami typu GLUT) a nabízí vysoký výkon, multiplatformnost a mnoho dalších výhod.
V případě, že zbudou nějaké časové prostředky, bude vyvinuta snaha renderer portovat i pro rozhraní DirectX (které je také velice rozšířené a používané) a v návrhu budu s touto možností počítat.

4.4 Výběr knihovny pro GUI

I když to není přímou součástí zadání, tak je plánováno pro program udělat grafické uživatelské rozhraní.
K tomuto účelu byla vybrána knihovnu wxWidgets [11].
Důvodem je zejména fakt, že mám s knihovnou již poměrně bohaté zkušenosti (jak ve škole, tak v běžné praxi). Knihovna má tyto kladné vlastnosti:

- stabilní knihovna pro tvorbu grafických uživatelských rozhraní
- multiplatformní — stačí sestavit s knihovnami pro daný operační systém
- velmi kvalitní dokumentace a uživatelská podpora
- jednoduché a intuitivní psaní aplikací
- licence umožňující z hlediska uživatele libovolné (i komerční) využití

4.5 Předpoklady pro návrh

V návrhu tedy bude zřetelná snaha o jednoznačné oddělení animačního a zobrazovacího (renderovacího) algoritmu.
Z hlediska uživatele bude od začátku bráno na zřetel, aby implementace poskytovala prostředky pro snadnou manipulaci parametrů simulace s ohledem na možný budoucí vývoj grafického uživatelského rozhraní (ať již jako součást práce nebo jako její rozšíření).

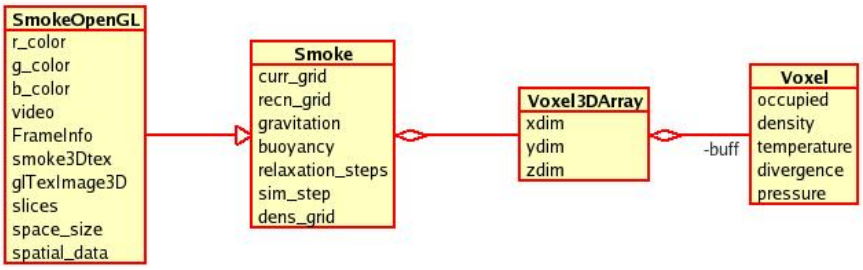
Kapitola 5

Návrh a implementace

Z hlediska vývoje aplikace je velice důležitou etapou již samotný návrh. Je vhodné použít objektového paradigmatu kvůli tomu, že návrh zpřehledňuje a umožňuje odhalit mnoho chyb ještě před vlastní implementací.
Z těchto důvodů bylo použito UML pro počáteční objektový návrh, který byl konzultován s vedoucím a až po malých úpravách (a validaci) bylo přikročeno k samotné implementaci.
Ke spojení návrhu a implementace bylo přistoupeno z důvodu velké provázanosti těchto dvou vývojových fází.
V této kapitole tedy vyrazíme z opačného konce, od implementace, a budeme ukazovat jak se fyzikální modely, popsané v kapitole 4 *Analýza*, diskretizují a jejich výpočty provádějí na počítačích. Tyto ukázky budou doplněny obrázky a kódem.

5.1 UML diagram a použité třídy

Na obrázku 5.1 můžeme vidět zjednodušený UML diagram vyvinuté aplikace.
Na první pohled je patrné, že množství použitých tříd není velké — to je způsobeno tím, že většina funckcionality se skrývá ve třídě *Smoke*, protože nebyl nalezen žádný pádný důvod pro dekompozici do menších celků.
Nyní se budeme věnovat popisu jednotlivých tříd a začneme od té nejzákladnější, kterou je *Voxel*.



Obrázek 5.1: Zjednodušený UML diagram aplikace.

5.1.1 Třída Voxel

Voxel je základním stavebním kamenem celé simulace. Modeluje jednu diskrétní část celého diskretizovaného prostoru a nese informace důležité pro výpočty spojené se simulací. Třída samotná je velice malá a neposkytuje žádnou složitou funkcionalitu. Mnohem důležitější je sémantika jejích jednotlivých atributů vzhledem k simulaci. Proto si uvedeme jejich seznam a roli ve výpočtech, které provází každý simulační krok naší metody.
Atributy třídy *Voxel*:

occupied (*bool*) Obsazenost voxelu pevným tělesem

density (*double*) Míra hustoty kouře (použito např. v rovnici 4.5)

temperature (*double*) Teplota vzduchu v daném bodě (taktéž použito v rovnici 4.5)

divergence (*double*) Laicky rozdílnost rychlostí v okolí voxelu — použito pro výpočet tlaku

pressure (*double*) Tlak způsobený prouděním vzduchu (společně s divergencí součást řešení Poissonovy rovnice)

forces (*{double, double , double}*) Externí síly — prostředek jak zasahovat do kouře (zde je možné aplikovat uživatelem definované síly, ale použito i pro aplikaci *Vorticity Confinementu* (sekce 4.1) nebo rovnice 4.5)

velocities (*{double, double , double}*) Rychlost proudění vypočtená advekcí (více sekce 4.1) a korigováná externími silami

5.1.2 Třída Voxel3DArray

Voxel3DArray ve své podstatě pouze zapouzdřuje pole prvků třídy *Voxel* a implementuje přístup k atributům za pomocí souřadnic v troj-rozměrném prostoru. Třída jako taková je implementována a oddělena od ostatních tříd proto, aby mohlia být řádně otestovat a aby bylo možné se později bez pochyb spolehnout na její služby.
Pouze pro úplnost informace bude uvden příklad kódu, s kterým se jednorozměrné pole (tak je fyzicky uloženo) adresuje jako trojrozměrné (za předpokladu, že známe všechny 3 rozměry prostoru).
Příklad adresace pole (v tomto případě zjišťujeme, zda je daný voxel obsazen):

```
bool Voxel3DArray::GetOccupiedAt(tDimension x, tDimension y, tDimension z)
{
        return buff[z*xdim*ydim + y*xdim + x].GetOccupied();
}
```

5.1.3 Třída Smoke

Smoke je nejdůležitější a z hlediska objemu i největší třídou celé aplikace. Udržuje v sobě veškeré atributy i funckcionalitu spojenou se simulací kouře (ale zcela záměrně neobsahuje nic, co by souviselo s vykreslováním — k tomu slouží třída *SmokeOpenGL*, která je potomkem třídy *Smoke*).
Podobně jako u třídy *Voxel* začneme výčtem, z hlediska simulace, důležitých atributů.
Atributy třídy *Smoke*:

curr grid (*Voxel3DArray*) Prostorová mřížka nesoucí informaci o aktuálním stavu výpočtu

recn grid (*Voxel3DArray*) Prostorová mřížka nesoucí informaci o předchozím stavu výpočtu

gravitation (*double*) Míra gravitace působící na kouř (dalo by se také popsat jako "težkost" kouře) — tento atribut v podstatě odpovídá proměnné *alpha* z rovnice 4.5 a tudíž je spojen s hustotou (*density*) kouře.

buoyancy (*double*) Míra vznosnosti působící na kouř — tento atribut v podstatě odpovídá proměnné *beta* z rovnice 4.5 a tudíž je spojen s teplotou (*temperature*) kouře.

timestep (*double*) Velikost časového kroku simulace — tento parametr hodně ovlivňuje kvalitu a rychlost simulace (viz. níže)

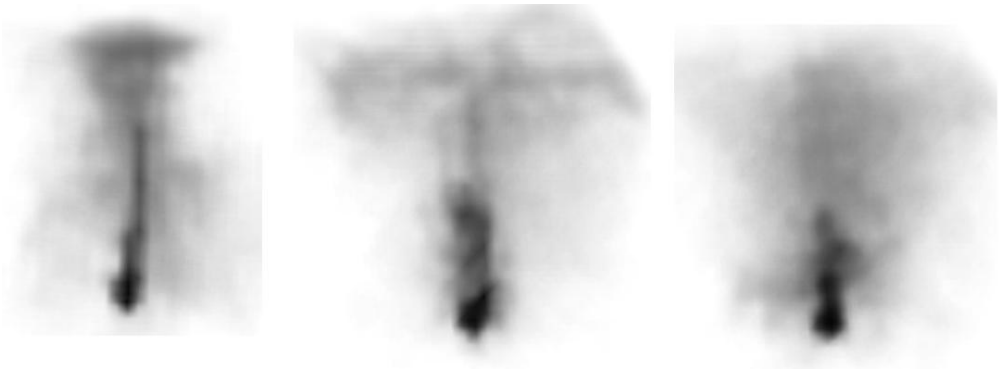
relaxation steps (*int*) Počet kroků pro *Gauss—Seidelovu* relaxační metodu — využito k převodu divergence na tlak

vorticity (*double*) Míra energie vrácené do modelu metodou *Vorticity Confinement* (viz. sekce 4.1)

Všechny simulační kroky provádí metoda *Smoke::Step()*, která je zároveň důležitým prvkem rozhraní mezi simulací a zobrazením. Pseudkód této metody a rozbor jednotlivých simulačních kroků bude uveden níže.

5.1.4 Třída SmokeOpenGL

SmokeOpenGL je přímým potomkem třídy *Smoke* a tvoří rozhraní mezi simulací a aplikací využívající OpenGL (v našem případě *Glut*). Tato třída se stará o správnou inicializaci OpenGL a poskytuje metodu *SmokeOpenGL::Draw()*, která pomocí *texture mappingu* (viz. kapitola 3 Zobrazení) vykresluje aktuální stav simulace do OpenGL. Ukázku takového vykreslování můžete vidět na obrázku 5.2. Při důkladném prozkoumání lze na prostředním obrázku pozorovat proužkovité artefakty charakteristické pro *texture mapping*.



Obrázek 5.2: Ukázka kouře vykresleného *texture mappingem*.

5.2 Pseudokód algoritmu a popis implementace jednotlivých kroků

V této velice důležité sekci uvedu pseudokód [1] simulačního algoritmu, který je srdcem celé diplomové práce. Hned po tom bude následovat popis jednotlivých kroků simulace (a hlavně způsob jejich implementace). Tento výsledný kód byl vytvořen na základě studia mnoha prací zabývajících se podobnou tématikou a zde je vhodné čtenáře odkázat minimálně na [4] a [12].
Normální výpočet probíhá pro všechny neokrajové voxely. Pro okrajové voxely se počítají okrajové podmínky (*Boundary Conditions*) — více viz. sekce Okrajové podmínky.

```
SwapGrids();
curr_grid->Clear();

FOR ALL VOXELS
// EXTERNAL FORCES PART

        BuoyancyForces();
        VorticityConfinementForces();
        ApplyForcesToVelocity();
        BoundaryConditions();

// MOMENTUM CONSERVATION PART

        VelocitySelfAdvection();
        BoundaryConditions();

// MASS CONSERVATION PART

        ComputeDivergence();
        ConvertDivergenceToPressure();
        BoundaryConditions();
        ApplyPressureToVelocity();
        BoundaryConditions();

// DENSITY AND TEMPERATURE ADVECTION PART

        AdvectDensityAlongVelocity();
        AdvectTemperatureAlongVelocity();
END
```

5.2.1 Vznosná síla (*Buoyancy*)

Vznosná síla je jednou z přirozených sil, která ovlivňuje chování kouře a je tudíž velice vhodné ji modelovat. Z hlediska chování kouře v realitě si každý znás může představit dva

[1] Pseudokód je v angličtině, protože bylo rozhodnuto, že je to přehlednější a přirozenější než při použití češtiny. Došlo také z jednodušení cyklů přes voxely na jediný (ve skutečnosti je jich mnohem víc — pro jednotlivé kroky výpočtu).

extrémy — horký a lehký, rychle stoupající kouř (cigaretový kouř), a težký a studený kouř klesající k zemi (mlha). Oba tyto jevy lze simulovat díky aplikaci vznosných sil (parametrů *alpha* a *beta* v rovnici 4.5).
Náznaky těchto simulací lze pozorovat na obrázcích 5.3 a 5.4;

Obrázek 5.3: Ukázka lehkého, stoupajícího kouře.

Obrázek 5.4: Ukázka těžkého, klesajícího kouře

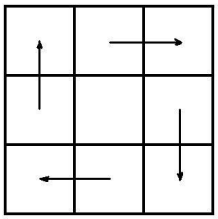
Implementace tohoto jevu je velice snadná — jde pouze o to určit oba parametry (zadá uživatel), vypočíst pro ně rovnici 4.5 a poté přidat vypočtenou sílu (která míří vzhůru) do modelu.
Kód, který je přímo použit v simulaci (opakuje se pro každý voxel) vypadá takto:

```
// computing buoyancy force and adding it to force field
tForce temp_buoyancy = - gravitation*recn_grid->GetDensityAt(x, y, z)
                       + buoyancy*(recn_grid->GetTemperatureAt(x, y, z)
                       - GetAmbientTemperatureAt(x, y, z));
curr_grid->AddYforceAt(x, y, z, temp_buoyancy);
```

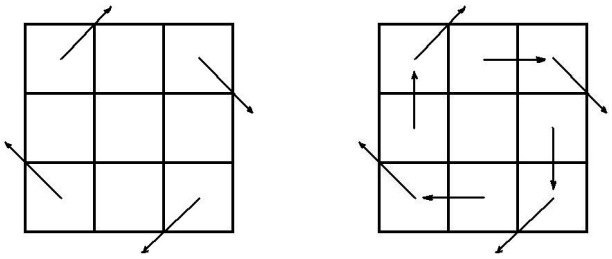
Proměnné *gravitation* a *buoyancy* jsou atributy třídy *Smoke* (viz. sekce 5.1).

5.2.2 Vorticity Confinement

Jak jsme již bylo popsáno výše, tak *Vorticity Confinement* je metoda představena Steinhoffem v [13]. Základní myšlenkou je vylepšení (nebo můžeme taky napsat opravení) malých turbulentních jevů v kouři, které z modelu mizí díky diskretizaci. Algoritmus zpočátku vypadal komplikovaně, ale nakonec jeho implementace byla poměrně jednouduchá a efektivní.
Pro každý voxel je tedy vypočtena jeho vířivost (*vorticity*) — ta je představována vektorem. Tento vektor je kolmý na rovinu víření a jeho velikost nám udává intenzitu víření.
Vektor víření je reprezentován stejně jako vektor síly nebo rychlosti — třemi složkami podél os dimenzí. Pro získání vektoru víření tedy stačí provést výpočet intenzity víření v každé dimenzionální rovinně a považovat jej za jednu ze tří složek vektoru.
Intenzita víření v jedné rovině se počítá podle okolních vektorů rychlosti (tyto jsou kolmé na aktuálně počítaný vektor). Na obrázku 5.5 vidíme jak se počítá intentzita víření v implementované aplikaci. Během implementace bylo zjištěno, že by se daly uplatnit i jiné formy



Obrázek 5.5: Matice pro výpočet vířivosti použitá v aplikaci.



Obrázek 5.6: Ukázka alternativních matic pro výpočet vířivosti.

těchto "filtračních matic" (ty jsou naznačeny na obrázku 5.6).

Získaný vektor je pak násoben proměnnou *Smoke::vorticity* a sílu, která tímto vznikne, je přidána do aktuálního pole externích sil — je distribuována zpět do okolních silových složek podle použitého modelu. Kód pro výpočet podle naznačeného modelu vypada takto [2]:

```
xvrtc = ( + recn_grid->GetZvelocityAt(x, y - 1, z)
          - recn_grid->GetZvelocityAt(x, y + 1, z)
          + recn_grid->GetYvelocityAt(x, y, z + 1)
          - recn_grid->GetYvelocityAt(x, y, z - 1))/timestep;
// X vorticity (normal vector)
curr_grid->AddYforceAt(x, y, z + 1, +xvrtc*vorticity);
curr_grid->AddYforceAt(x, y, z - 1, -xvrtc*vorticity);
curr_grid->AddZforceAt(x, y - 1, z, +xvrtc*vorticity);
curr_grid->AddZforceAt(x, y + 1, z, -xvrtc*vorticity);
```

Jak vidíme, tak implementace *Vorticity Confinementu* není příliš náročná — ani z hlediska programátorského usílí, ani z hlediska rychlosti běhu programu. Což je důležité, protože tato metoda dokáže velice výrazně pozvednout vizuální kvalitu výsledného efektu (samozřejmě v závislosti na vhodně zvolené proměnné *Smoke::vorticity*). Na obrázcích 5.8 a 5.7 si

[2] Zde pouze pro jednu dimenzinální rovinnu kvůli úspoře místa

můžete prohlédnout dva kouřové efekty, které se svými parametry liší pouze v použití (resp. nepoužití) metody *Vorticity Confinement*.

Obrázek 5.7: Ukázka kouře bez použití *Vorticity Confinementu*.

Obrázek 5.8: Ukázka kouře s použitím *Vorticity Confinementu* (jinak má stejné parametry jako obrázek 5.7).

5.2.3 Aplikace externích sil (*External forces*)

Aplikace externích sil je triviální záležitostí. Všechny silové vektory se pouze vynásobí časovým krokem a přidají k aktuálnímu poli rychlostních vektorů.
Kód vypadá následovně:

```
vlct = timestep*curr_grid->GetXforceAt(x, y, z);
curr_grid->AddXvelocityAt(x, y, z, vlct);

vlct = timestep*curr_grid->GetYforceAt(x, y, z);
curr_grid->AddYvelocityAt(x, y, z, vlct);

vlct = timestep*curr_grid->GetZforceAt(x, y, z);
curr_grid->AddZvelocityAt(x, y, z, vlct);
```

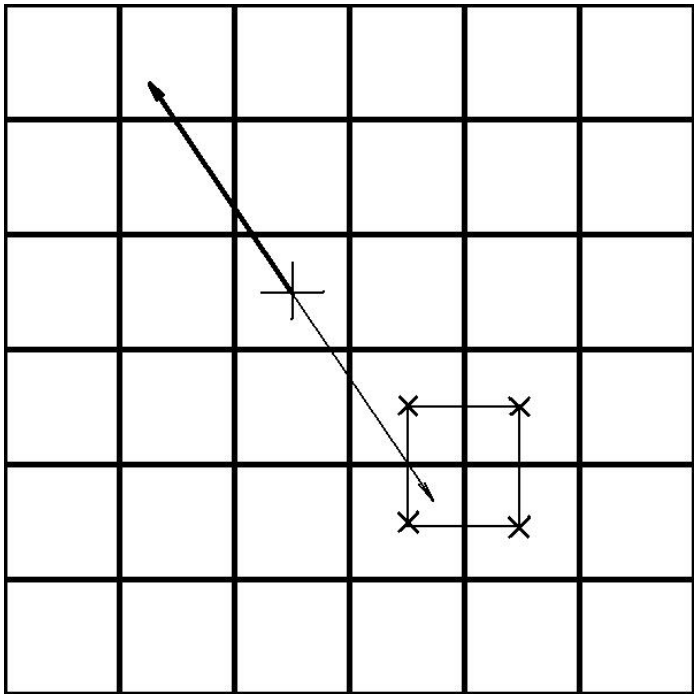
Jak již bylo uvedeno, tak se touto formou do modelu přidávají externí síly. V případě našeho modelu to znamená uživatelské síly, vznosné síly a síly, které vzniknou metodou *Vorticity Confinement*.
Na obrázku 5.9 můžeme pozorovat rozptyl kouře do stran díky uživatelské síle působící směrem dolů.

5.2.4 Sebe—advekce rychlosti (*Velocity self—advection*)

Sebe—advekce rychlosti je esenciální složkou celého algoritmu. K tomuto kroku lze přistupovat mnoha způsoby, ale byl vybrán způsob uvedený v [12].
Tato metoda je implicitní a bezpodmínečně stabilní (narozdíl od jiných metod), což je pro

Obrázek 5.9: Ukázka použití externí uživatelské síly (působící uprostřed prostoru směrem dolu).

potřeby simulace nezbytné.
Hlavní myšlenkou je zpětné vyhledávání (*backtracking*) rychlostních složek z minulého kroku simulace (oproti dopředné distribuci rychlosti v implicitních metodách).
Pro každý voxel je tedy vypočítán bod, ze kterého v minulém kroku vyšel a jsou přeneseny rychlostní složky z tohoto bodu do aktuálního rychlostního pole. Tento složitý popis by se měl trochu vyjasnit po shlédnutí obrázku 5.10. Bod získaný zpětným vyhledáním ve většině případů samozřejmě nepadne doprostřed nějakého voxelu, a proto je nutné rychlostní složky v tomto bodě interpolovat z okolí. Prostorové interpolaci se budeme věnovat v následující sekci.
Další osvětlení problému může přinést tento pseudo—kód [3]:



Obrázek 5.10: Ukázka použití externí uživatelské síly (působící uprostřed prostoru směrem dolu). Silnější šipka ukazuje původní směr rychlostního vektoru, ale my potřebujeme přesně opačný vektor (tenčí šipka) na jehož konci interpolujeme rychlost s naznačených bodů.

```
// computing distances of backtraced point
```

[3] Pseudo—kód proto, že bylo vynecháno ošetření přesažení simulačního prostoru.

```
xdst = - timestep*recn_grid->GetXvelocityAt(x, y, z);
ydst = - timestep*recn_grid->GetYvelocityAt(x, y, z);
zdst = - timestep*recn_grid->GetZvelocityAt(x, y, z);

// getting coordinates of tracked point
xcrd = x + xdst;
ycrd = y + ydst;
zcrd = z + zdst;

curr_grid->AddXvelocityAt(x, y, z, GetInterpoledXVelocityAt(xcrd, ycrd, zcrd));
curr_grid->AddYvelocityAt(x, y, z, GetInterpoledYVelocityAt(xcrd, ycrd, zcrd));
curr_grid->AddZvelocityAt(x, y, z, GetInterpoledZVelocityAt(xcrd, ycrd, zcrd));
```

Vpodstatě je tedy pouze převrácen vektor rychlosti (z minulého kroku) opačným směrem a rychlost, interpolovanou v bodě kam tento vektor ukazuje, je přidána do aktuálního rychlostního pole. V tomto místě je třeba zmínit, že je použit nejjednodušší způsob intergrace pro získaní interpolačního bodu — je to z důvodu úspory jak výpočetního, tak programátorského času. Pro realističtější simulaci (která však za cenu rychlosti není potřeba) by zde bylo vhodné použit nějakou sofistikovanější metodu integrace (např. *Semi-Lagrangevou metodu — popsáno např. v [4]*).

5.2.5 Prostorová interpolace

Z důvodů uvedených v předchozí sekci je nutné interpolovat rychlostní vektor v libovolném bodě simulačního prostoru. Pro tento účel byla zvolena nejjednodušší lineární interpolac.
Kód interpolace vypadá následovně:

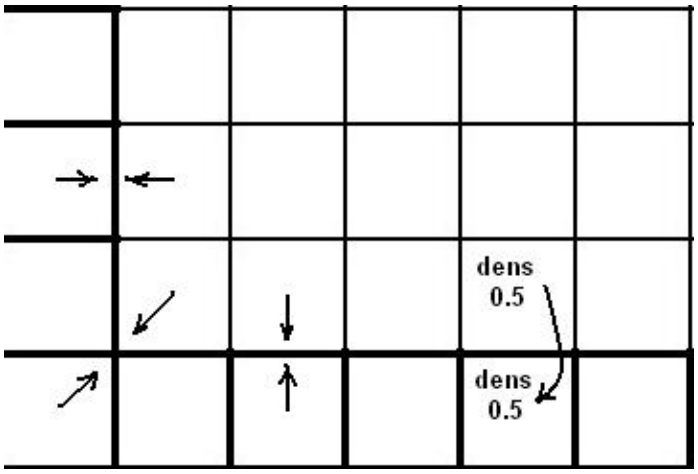
```
tRealValue Smoke::TriLinearInterpolation(tRealValue a, tRealValue b,
                                         tRealValue c, t8values v)
{
return + (1.0 - a)*( (1.0 - b)*( (1.0 - c)*v.v000
       + c*v.v001 ) + b*( (1.0 - c)*v.v010 + c*v.v011 ) )
       + a*( (1.0 - b)*( (1.0 - c)*v.v100 + c*v.v101  )
       + b*( (1.0 - c)*v.v110 + c*v.v111 ) );
}
```

Parametry a, b a c jsou souřadnice v interpolovaném prostoru (tedy ne souřadnice v celém simulačním prostoru, ale pouze už v konkrétním voxelu) a parametr v je struktura osmi okolních hodnot. Použitý výpočetní vzorec je standardním řešením lineární prostorové interpolace.
Zde je opět potřeba zmínit, že použití lepšího interpolačního schématu by vedlo ke zvýšení realističnosti (čtenář je odkázán například na [4], kde je navržen zajímavý monotonní kubický interpolator).

5.2.6 Okrajové podmínky (*Boundary conditions*)

Nastavení vhodných okrajových podmínek je důležité pro správné chování kouře při okrajích simulačního prostoru (ten je v našem případě poměrně dost omezený, a proto nastavení těchto podmínek nemůžeme zanedbat). V našem případě jde o to, aby rychlost (ale i ostatní

veličiny) byla u okrajů upravena takovým způsobem, aby nemířila ven ze simulačního prostoru.
Tato úloha není příliš složitá pokud je brán v úvahu 2-D prostor. Na obrázku 5.11 jsou zobrazeny hraniční stavy a k nim vztažené okrajové podmínky. V případě 2-D prostoru tedy existuje celkem 8 okrajových podmínek (4 ohraničující strany + 4 rohy). Ve všech těchto situacích (tedy na těchto pozicích v prostoru) je třeba odlišným způsobem nastavit chování simulačních veličin.
Pro 3-D prostor ovšem počet těchto podmínek výrazně stoupá na konečnou sumu 26 (8

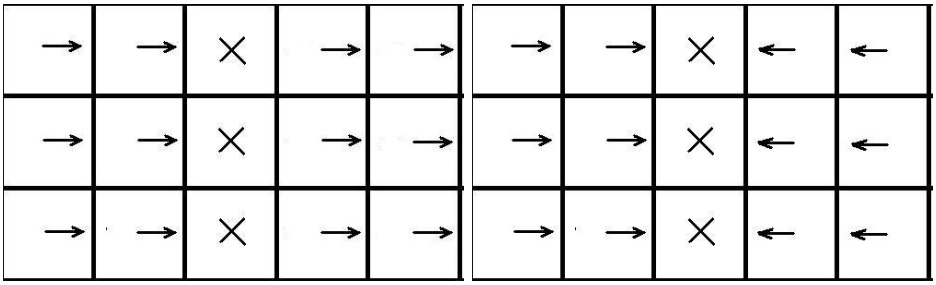


Obrázek 5.11: Ošetření okrajových podmínek. Šipkami jsou naznačeny okrajové rychlostní složky, které "tlačí" rychlostní vektory zpět do simulačního prostoru (resp. nulují jejich složky mířící ven). Je také ukázáno použití kontinuality pro skalární veličiny (v tomto případě hustota kouře).

rohů + 6 okrajových ploch + 12 hran). I přesto bylo rozhodnuto všechny tyto podmínky implementovat, protože jsou považovány za důležitou součást aplikace.
U všech veličin, kromě těch vektorových (tedy rychlost a tlak), se předpokládá pouze kontinualita. Tedy například u hraničních ploch se tato veličina přenese (okopíruje) z nejbližšího neokrajového voxelu. U vektorových veličin jde vždy o to, aby se proti jeho složce mířící ven ze simulačního prostoru postavila složka rušící tento jev.
Samozřejmě by bylo možné vymyslet i jiné okrajové podmínky — například rychlost, která končí na jedné straně prostoru by se objevila na protilehlé stěně (zde je vhodné se inspirovat např. v [12]).

5.2.7 Divergence

Divergence je prvním krokem implementace výpočtu Poissonovi rovnice. Vpodstatě jde pouze o rozdílnost rychlostních vektorů, která pak vede ke vzniku tlaku v tektuině. Pokud tedy tekutina (v našem případě vzduch) proudí jedním směrem a nenaráží na žadnou překážku (ať již pevnou nebo v podobě tekutiny s jiným vektorem rychlosti), tak je její divergence (a tudíž i tlak) nulový. Pokud se srazí dva protichůdné proudy (nebo proud narazí na pevnou překážků), tak divergence v bodě srážky vroste (a s ní následně i tlak v okolí) a donutí tak tekutinu k tomu, aby se rozptylovala do stran. Na obrázku 5.12 jsou tyto dvě situace naznačeny.
Kód pro výpočet divergence vypadá následujícím způsobem:



Obrázek 5.12: Oba extrémní stavy v tekutině a jejich vliv na divergenci. Vpravo je plynulé proudění v jehož objemu nevzniká divergence. Vlevo jsou dva protichůdné proudy na jejichž rozhraní vzniká vysoká divergence. (místa týkající se divergence jsou označena křížkem)

```
curr_grid->SetDivergenceAt(x, y, z,
-0.5*(1/((tRealValue) xdim))*(curr_grid->GetXvelocityAt(x + 1, y, z)
                              - curr_grid->GetXvelocityAt(x - 1, y, z))
-0.5*(1/((tRealValue) ydim))*(curr_grid->GetYvelocityAt(x, y + 1, z)
                              - curr_grid->GetYvelocityAt(x, y - 1, z))
-0.5*(1/((tRealValue) zdim))*(curr_grid->GetZvelocityAt(x, y, z + 1)
                              - curr_grid->GetZvelocityAt(x, y, z - 1))
);
```

5.2.8 Převod divergence na tlak

Jak již bylo předesláno v minulé sekci, tak je potřeba převést divergenci na tlakové pole. V tomto ohledu je postup opět inspirován článkem [4] a zejména [12]. V obou pracech je použita jednoduchá *Gauss—Seidelova* relaxační, iterační metoda. Její implementace není příliš náročná a poskytuje spolehlivé a konvergující výsledky. V kapitole 6 *Výkon* bude tato metoda podrobena měření, které by mělo ukázat do jaké míry zaťežuje výpočet svou časovou náročností. Implicitně je v implementované aplikaci použito 20 iterací pro převod divergence na tlak, ale tuto hodnotu lze měnit a experimentovat s ní (i za běhu programu).
Následuje pseudo—kód *Gauss—Seidelovy* metody implementovaný v aplikaci:

```
FOR RELAXATION_STEPS
        curr_grid->SetPressureAt(x, y, z,
            (curr_grid->GetDivergenceAt(x, y, z) +
            curr_grid->GetPressureAt(x + 1, y, z) +
            curr_grid->GetPressureAt(x - 1, y, z) +
            curr_grid->GetPressureAt(x, y + 1, z) +
            curr_grid->GetPressureAt(x, y - 1, z) +
            curr_grid->GetPressureAt(x, y, z + 1) +
            curr_grid->GetPressureAt(x, y, z - 1)
        ) / 6.0
        );
END
```

5.2.9 Aplikace tlaku na rychlostní pole

V tento moment je potřeba aplikovat vypočítaný tlak na rychlostní pole. To se děje průměrováním tlakového vektoru. Je ještě potřeba jej převézt zpět z normalizované podoby vynásobením rozměrem prostoru.
Kód, který tento krok implementuje, vypadá takto:

```
curr_grid->AddXvelocityAt(x, y, z,
        -0.5*(curr_grid->GetPressureAt(x + 1, y, z)
              - curr_grid->GetPressureAt(x - 1, y, z))
        *xdim);
curr_grid->AddYvelocityAt(x, y, z,
        -0.5*(curr_grid->GetPressureAt(x, y + 1, z)
              - curr_grid->GetPressureAt(x, y - 1, z))*ydim);
curr_grid->AddZvelocityAt(x, y, z,
        -0.5*(curr_grid->GetPressureAt(x, y, z + 1)
              - curr_grid->GetPressureAt(x, y, z - 1))*zdim);
```

Po tomto kroku je tedy rychlostní pole v konečné podobě a splňuje rovnice 4.1 a 4.2.

5.2.10 Advekce teploty a hustoty

V závěru celého výpočetního kroku už zbývá pouze posunout teplotu a hustotu kouře (ta nás zejména zajímá, protože je vykreslována) podle vypočteného rychlostního pole. Tento krok je velice analogický k advekci rychlosti, což je patrné z následujícího kódu [4]:

```
xdst = - timestep*curr_grid->GetXvelocityAt(x, y, z);
ydst = - timestep*curr_grid->GetYvelocityAt(x, y, z);
zdst = - timestep*curr_grid->GetZvelocityAt(x, y, z);

// getting coordinates of tracked point
xcoord = x + xdst;
ycoord = y + ydst;
zcoord = z + zdst;

curr_grid->AddDensityAt(x, y, z,
        GetInterpoledDensityAt(xcoord, ycoord, zcoord));
curr_grid->AddTemperatureAt(x, y, z,
        GetInterpoledTemperatureAt(xcoord, ycoord, zcoord));
```

5.3 Rozhraní

V této sekci dojde k popisu rozhraní tříd *Smoke* a *SmokeOpenGL*, tak aby bylo možné bez problémů vyvinout jinou nadstavbu aplikace (ať již vykreslování objemových dat nebo GUI rozhraní apod.). Při tvorbě aplikace byla vyvinuta snaha o to ji udělat co nejpřehlednější a příznivou pro budoucí úpravy, takže to nebude těžký úkol.

[4] Opět je vynecháno ošetření na přesažení okraje

5.3.1 Rozhraní třídy *Smoke*

Konstruktor třídy *Smoke* existuje pouze v podobě bez parametrů. Incializaci dat obstarává funkce *CreateGrids(long, long, long)*, která vytvoří simulační prostory. Výpočetní krok simulace je proveden voláním funkce *Step()* (jejíž obsah jsme si podrobně popsali v minulé sekci).
Příklad použití těchto metod:

```
Smoke * smoke = new Smoke();
smoke->CreateGrids(30, 30, 30);
while(!end_simulation)
{
    smoke->Step();
}
```

Toto byly tedy úplně základní prvky ovládání průběhu animace. Teď si popíšeme jak získat informace o simulaci, nebo jak upravit její parametry.
Rozměry inicializovaného prostoru získáme metodami *GetXdim*, *GetYdim* a *GetZdim*.
Gravitaci a vznosnost získáme/nastavíme metodami *{Get|Set}Gravitation()* a *{Get|Set}Buoyancy()*.
Časový krok získáme/nastavíme metodami *{Get|Set}TimeStep()*.
Míru energie vrácené algoritmem *Vorticity Confinement* získáme/nastavíme metodami *{Get|Set}Vorticity()*.
Počet kroků *Gauss—Seidelovy* metody získáme/nastavíme metodami *{Get|Set}RelaxationSteps()*.
Operace s hustotou a teplotou se provádějí metodami *{Add|Get|Set}DensityAt()* a *{Add|Get|Set}TemperatureAt()*.
Nakonec už zbývají pouze prostředky pro přidávání externích sil. O to se stará celkem 9 metod ve tvaru *{Add|Get|Set}{X|Y|Z}forceAt()*.
U všech metody končících suffixem *At* (např. *SetYforceAt()*) první tři parametry určují voxel v prostoru.
To je celé rozhraní třídy *Smoke*.
Teď je vhodné uvést příklad použití těchto metod:

```
Smoke * smoke = new Smoke();
smoke->CreateGrids(30, 30, 30);

smoke->SetGravitation(0.1);
smoke->SetBuoyancy(0.6);
smoke->SetTimeStep(0.1);
smoke->SetVorticity(0.01);

while(!end_simulation)
{
    for (int x = smoke->GetXdim()/2 -1; x <= smoke->GetXdim()/2 +1; x++)
    {
        for (int z = smoke->GetZdim()/2 -1; z <= smoke->GetZdim()/2 +1; z++)
```

```
        {
            for (int y = 3; y <= 3; y++)
            {
             smoke->AddYforceAt(x, y - 1, z, 5.0);
             smoke->SetTemperatureAt(x, y, z, 1.0);
             smoke->SetDensityAt(x, y, z, 1.0);
            }
        }
    }
    smoke->Step();
}
```

Tímto kódem provedeme simulaci lehkého, mírně turbulentního kouře a v každém kroku přidáme na dno doprostřed simulačního prostoru čtverec s teplotou, hustotou a silou tlačící vzduch vzhůru.
Z rozhraní je tedy patrné, že tato třída se absolutně vůbec nestará o vykreslování (pouze metodou *GetDensityAt()* umožňuje přístup k datům, která mohou sloužit pro vykreslování).

5.3.2 Rozhraní třídy *SmokeOpenGL*

Třídá *SmokeOpenGL* dědí z třídy *Smoke* a doplňuje ji o funckionalitu, která umožňuje vykreslovat simulační (objemová) data do okna OpenGL.
Metodou *Init(long, long, long)* se třída incializuje na daný prostor a také se provedou některá nastavení OpenGL (například získání adresy funkce *glTexImage3D*). Výpočetní krok se provádí schodně jako u třídy *Smoke*, tedy metodou *Step()*, která v sobě volá stejnojmenou metodu svého rodiče a přidává k ní manipulaci s daty a ukládání do videa. Pro vykreslení simulačního prostoru slouží funkce *Draw()*, ve které se tak skrývá většina funkcionality.
Pro ukládání videa slouží funkce *StartRecording()*, která za své parametry bere název výstupního souboru, rozměry videa a počet obrázku za sekundu (použití bude demonstrováno v krátkém příkladu). Ukládání videa se zastaví metodou *StopRecording()* — to je důležité vždy provést, jinak se video neuloží korektně.
Jako parametry pro vykreslovaní lze pouze nastavit počet řezů pro renderovací metodu *Texture Mapping* (sekce 3.2) a barvu kouře. To se provede metodami *SetSlices()* a *Set{R|G|B}Color()*.
Následuje příklad použití třídy *SmokeOpenGL*:

```
SmokeOpenGL * smoke = new SmokeOpenGL();
smoke->Init(60, 60, 60);

// je vhodne nastavit rozmery stejne jako ma okno GLUTu
smoke->StartRecording("out.avi", 800, 600, 25);

// GLUT DisplayFuncCallback
void display(void)
{
    . . .

    // pouzita varianta funkce Draw pro rotaci kolem dvou os
```

```
    smoke->Draw(((float)xnew), ((float)ynew));
    glutSwapBuffers();                          /* a prohozeni bufferu */
}

// GLUT TimerFuncCallback - volano periodicky GLUTem
void timer(void)
{
    . . .

    smoke->Step();

    . . .
    glutPostRedisplay(); \\ vykresleni vysledku simulacniho kroku
}

// provezt na konci funkce main()
smoke->StopRecording();
```

K implementaci je potřeba uvést, že pro vykreslování si třída interně neuchovává buffer o stejných rozměrech jako simulační prostor. Udržuje se prostor ve tvaru krychle, jejíž strana má vždy rozměr o velikosti mocniny dvou (větší než největší hrana simulačního prostoru). Bylo totiž zjištěno, že OpenGL krychli o jiných rozměrech zobrazuje výrazně pomaleji (nejspíše z důvodu nějakých optimalizací). Při vykreslování je tedy poměrně komplikovaným způsobem převáděn simulační prostor na vykreslovací prostor. Nicméně tento postup funguje a umožňuje uživateli simulovat prostory o libovolných rozměrech (samozřejmě s ohledem na výkonost).

5.4 Výsledná aplikace a postupy a nástroje použité při jejím vývoji

Výsledná aplikace byla implemetována na platformě *Windows* s překladačem *MinGW*. Pro vývoj nebylo použito žádného vývojového prostředí — byl použit pouze uvedený překladač a editor textů *EditPlus*. Při tvorbě obslužné aplikace byla použita knihovna *GLUT*.
Nakonec bylo učiněno rozhodnutízůstat u principu jednoduché konzolové aplikace, u které je možné modifikovat proměnné simulace jak z příkazové řádky (pomocí argumentů programu)tak stiskem kláves za běhu programu.

5.4.1 Platforma Windows

Vývoj aplikace byl započat na Linuxu, na školním serveru *merlin*. Během prací se ale ukázala potřeba pokračovat v dalším vývoji na Windows. Jediným důvodem byla možnost pracovat na projektu i na cizích počítačích (bez Linuxu a připojení k internetu). Nicméně není složité přeportovat projekt i do Linuxu. Z hlediska vývoje se se zvolenou platformou nevyskytly žádné problémy, ale asi by bylo zajímavé srovnat ostatní platformy z hlediska výkonosti.

5.4.2 Překladač MinGW

MinGW je sada překladačů a nástrojů pro vývoj programů na platformě Windows. Jako taková není distribuována pod licencí *GPL*, takže je možné vyvinuté aplikace distribuovat bez poskytnutí zdrojového kódu (je pozitivní, že tuto možnost oproti jiným knihovnám máme). Rozhodnutí použít tuto knihovnu bylo provedeno na základě dobrých zkušeností s jejím použitím na jiných projektech.

5.4.3 Editor textů EditPlus

EditPlus je sharewarovým nástrojem zaměřeným na editaci zdrojových textů mnoha různých jazyků. Podporuje zvýraznění syntaxe a automatickou tvorbu častých programových konstrukcí. Z hlediska vývoje projektu je velice důležitý systém vyhledávání a nahrazování používající regulární výrazy a umožňující pracovat nad více soubory.

5.4.4 GLUT — nadstavba OpenGL

GLUT je nadstavbou OpenGL, která umožňuje snadnější tvorbu programů. Poskytuje multi-platformní řešení pro zobrazování oken, obsluhu uživatelských vstupů (klávesnice a myš) a další pomůcky, které výrazně usnadňují práci s OpenGL. Funkce GLUTu lze od běžného OpenGL kódu rozeznat pomocí prefixu *glut*.

5.4.5 Aplikace

Jak již bylo zmíněno, tak aplikace je pojata jako konzolová. Tvorba GUI nebyla součástí zadání, a proto bylo nakonec rozhodnuto aplikaci ponechat v této podobě. GUI by sice zpřehlednilo a zpříjemnilo použití aplikace, ale na druhou stranu je v současném stavu možné aplikaci ovládat skriptem, čehož bylo využito například při testování výkonu (viz. kaptila 6 *Výkon*).
Všechny ovládací prvky (a ostatní informace) aplikace vypisuje na standartní výstup s jehož pomocí dokáže program ovládat i nový uživatel.
Aplikace automaticky ukládá videozáznam průběhu simulace — název výstupního souboru lze specifikovat jedním z přepínačů programu. Výstup do videa je nezbytný pro sledování animací na větších simulačních prostorech, protože potřebný výpočetní výkon kubicky roste se stranou simulačního prostoru (v kapitole 6 *Výkon* je tato závislost popsána). Simulační krok pak může trvat i několik desítek sekund a zcela se tak potírá vjem dynamických vlastností simulace. Pro výstup do videa je použita třída *AVFile* poskytnutá vedoucím práce.
Pro detailnější popis aplikace a jejího ovládání je doporučeno prozkoumat přílohu A *Ovládání aplikace a ukázky výstupu*.

Kapitola 6

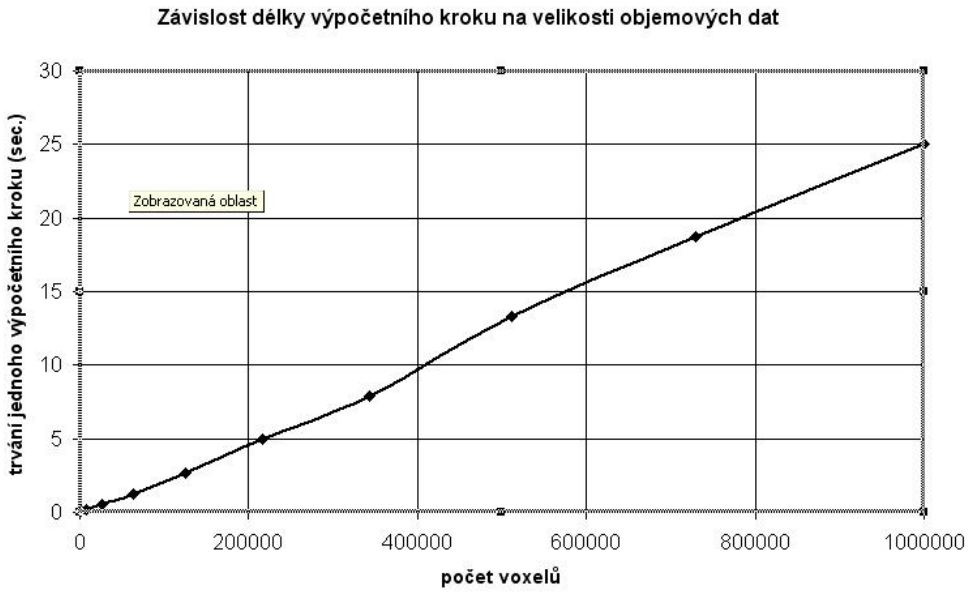
Výkon

Tato kapitola se zaměří na prozkoumání výkonu aplikace ze dvou různých pohledů.
V první bude provedeno měření délky jednotlivých simulačních kroků v závislosti na velikosti simulačního prostoru a bude potvrzena (intuitivně pomocí grafu) domněnka, že doba trvání jednoho kroku je přímo úměrná velikosti simulačního prostoru. V této části bude také provedeno výkonostní srovnání vytvořené aplikace s aplikací vytvořenou v [4].
V druhé části bude provedeno měření procentuální délky trvání jednotlivých fází kroků (ty odpovídají sekci 5.2 *Pseudokód algoritmu*). Na základě zjištěných kroků bude provedena krátká analýza, ve které budou označena (z hlediska výkonu) "slabá" místá implementace a bude učiněn pokus navrhnout alespoň částečné řešení (buď formou použití jiné metody nebo optimalizací stávajícího kódu).

6.1 Absolutní výkon aplikace — délka kroků v závislosti na velikosti simulačního prostoru

Pro účely měření byly do kódu vloženy podmíněné výpočty a výstupy simulačních statistik. Poté byl vytvořen skript, který pro všechny měřené prostory (kubické, velikost hrany 10 až 100 voxelů) provedl měření (pro každý prostor 10 měření) a uložil výstupy do souboru. Tento soubor byl převeden do formátu *CSV (comma separated values)*, tak aby jej bylo možno načíst a zpracovávat v běžných tabulkových procesorech.
Pro každou velikost prostoru byla spočítána průměrná délka trvání jednoho simulačního kroku a tyto hodnoty byly vyneseny do grafu závislosti. Ten je možné si prohlédnout na obrázku 6.1.

Z grafu se tedy potvrzuje domněnka, že je tato závislost lineární, což nám vytváří prostor pro srovnání naší metody s jinými metodami.
Například simulace kubického prostoru (z [4]) o straně 40 voxelů (velikost prostoru 64000 voxelů) trvala na platformě *dual Pentium-3 800 Mhz* necelou jednu sekundu. Vyvinutá simulace provede odbdobný výpočet za 1.3 sekundy na platformě *AMD Athlon 64 3200+*. Pokud srovnáme větší prostor, například 100x100x40 (tedy 400000 voxelů), tak zjistíme, že je náš simulátor již rychlejší (10 sekund oproti 30-ti). Tato tendence (tedy poměr jedna ku třem) pro větší velikosti prostoru pokračuje (pro ovření srovnejte graf 6.1 s [4]).
Náš algoritmus je tedy srovnatelně rychlý s již vyvinutými, ale je potřeba brát v úvahu, že v našem případě simulujeme na pokročilejším (jak taktovací frekvencí, tak architekturou) procesoru. Je také třeba brát v potaz, že naše implementace je velice jednoduchá — používá

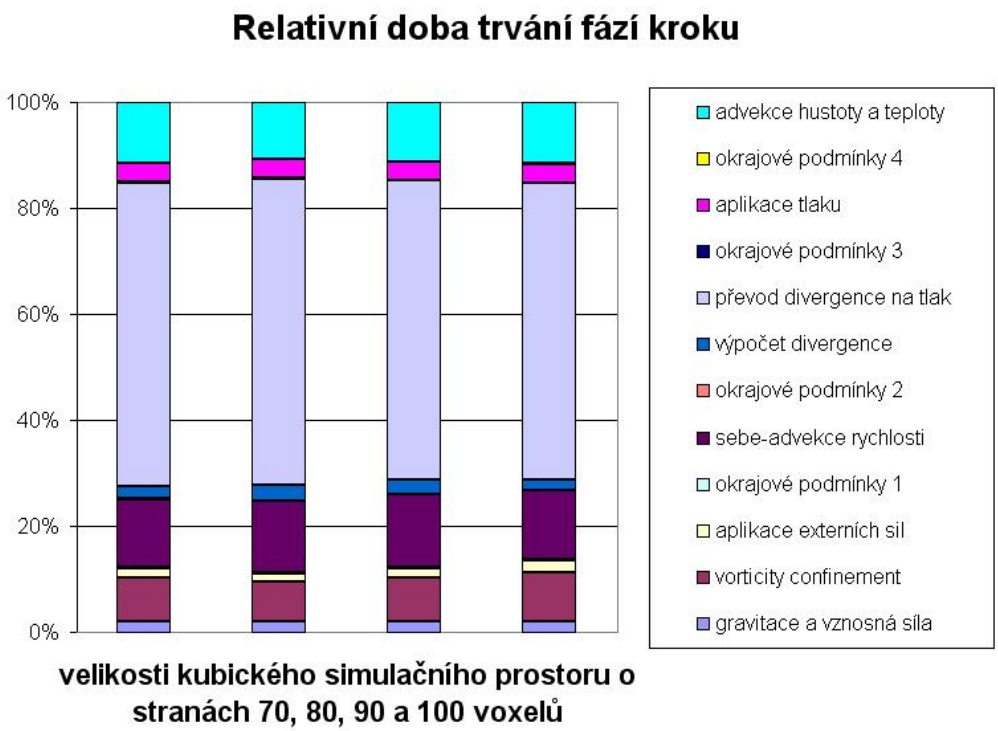


Obrázek 6.1: Graf závislosti délky trvání výpočetního kroku na velikosti simulačního prostoru.

nejjednodušší metody integrace a interpolace.

6.2 Relativní rychlost jednotlivých fází výpočetního kroku

Data pro měření byla získána obdobným způsobem jako u předchozí sekce. Výsledný graf byl vytvořen pro 4 největší simulační prostory, aby se ukázalo, zda má jejich velikost vliv na procentuální zastoupení jednotlivých fází v kroku.
Na obrázku 6.2 můžeme tedy sledovat, že velikost prostoru nemá nějaký zásadní vliv na tuto vlastnost.
Z grafu lze tedy odečíst některé důležité informace.
Výpočet okrajových podmínek (celkem 4-krát během výpočtu) tvoří zanedbatelnou složku celkového času (asi 4 %) a tudíž se nemusíme zabývat jejich optimalizací.
Aplikace externích sil a tlaku tvoří dohromady cca. 6 % celkového času a jejich optimalizace je takéž zbytečná (a vpodstatě nemožná — implementace je velice jednoduchá a další zjednodušení nebo zrychlení není takřka možné).
Vpodstatě totéž se dá napsat o výpočtu gravitačních (a vznosných sil) a divergence (celkem cca. 4 %).
Asi 10 % času zabere výpočet sil metody *Vorticity Confinement*. Tady by bylo možné drobných optimalizací dosáhnout s použitím jiných matic pro výpočet síly, ale k žádné výrazné úspoře již asi dojít nemůže. Je důležité si uvědomit (tak jak jsem to již zdůraznil v sekci 5.2), že tato metoda výrazným způsobem vylepšuje výslednou kvalitu simulace. Vypuštění této metody za účelem zrychlení tedy nepřichází v úvahu, protože její přínos mnohonásobně převyšuje využitý výkon.
Čtvrtinu času simulačního kroku spotřebují advekce (rychlosti, hustoty a teploty). Tento údaj je pochopitelný vzhledem k faktu, že tyto kroky používají lineární interpolaci. Hlavní



Obrázek 6.2: Graf procentuálního zastoupení fází ve výpočetním kroku.

prostor pro ovlivnění rychlosti tedy vidím v experimentování i s jinými druhy interpolace. Největší část výkonu (55 %) spotřebuje převod divergence na tlakové pole. Což je opět snadno pochopitelné kvůli použití relaxační metody (implicitně 20 průchodů). Zde se tedy opět nabízí možnost experimentovat s jinými relaxačními metodami a počtem relaxačních kroků.
Obecně by bylo možné některé výpočty dál optimalizovat přepisem násobení a dělení na bitové operace, ale z časových důvodů tato možnost nebyla ověřena.

Kapitola 7

Závěr

Hlavním úkolem celé diplomové práce bylo nastudování různých přístupů k animaci kouře a jeho zobrazení na jejichž základě měla být vytvořena aplikace implementující tyto metody. V uvedené literatuře jsou zmíněny všechny hlavní zdroje, z kterých bylo čerpáno (zde bych rád zmínil zejména [4]).
Během studia materiálů byla vytvořena konkrétní představa o současném stavu vývoje algoritmů pro realistickou simulaci tekutin. Byly pochopeny potřebné matematicko-fyzikální formulace, o které se tyto metody opírají. Bylo provedeno seznámení se s diskretizací těchto analytických prostředků a s různými formami optimalizace rychlosti a kvality zobrazení.
Došlo k prozkoumání současných přístupů k zobrazovaní volumetrických dat. Byl kladen důraz zejména na možnost využití moderních grafických akcelerátorů za účelem zrychlení vykreslování a zjednodušení implementace.
V obou oblastech (animace a zobrazení) byla provedena analýza dostupných algoritmů a byly vybrány (z pohledu vyvíjené aplikace) nejlepší přístupy, které byly využity při implementaci. Byly vybrány nástroje (programovací jazyk a grafická knihovna) vhodné pro vývoj. Algoritmy, které byly zvoleny, jsou již popsány a hlavním cílem celé diplomové práce byla jejich implementace.
Na základě nastudovaných materiálu byla navržena a vyvinuta aplikace pro realistickou animaci kouře. Při implementaci byla projevena snaha, aby byl program co nejvíce otevřen možným úpravám a aby se simulace (a zobrazení) daly jednoduše použít i v jiném programu. Aplikaci se úspěšně podařilo implementovat a její výstupy byly použity a prezentovány v rámci celé diplomové práce (zejména sekce 5.2).
Práce jako taková má mnoho možností jak může být rozšířena. Z hlediska techologií počítačové grafiky se nabízí zejména vývoj rendereru, který by data zobrazoval pomocí *Volume Ray-Castingu* (ten je popsán v sekci 3.1 a zajímavé řešení je nastíněno v [4]). Tím by se dosahlo zlepšení výsledné vizuální kvality kouře, ale na druhou stranu je tato metoda výpočetně náročnější než použitý *Texture Mapping* (sekce 3.2).
Dalším možným rozšířením je zdokonalení numerických metod použitých při simulace. Zde jsou myšleny zejména lepší metody integrace a interpolace při výpočtu advekce fyzikálních veličin (sekce 5.2). To by vedlo ke zlepšení míry realističnosti simulace, ale opět by došlo ke spotřebě většího množství výpočetního výkonu.
Také by bylo vhodné pokud by byla aplikace vybavena grafickým uživatelským rozhraním, které by učinilo ovládání aplikace jedodušším a intuitivnějším.
Tato rozšíření nebyla považována za součást řešení této práce, ale byl brán ohled na to, aby integrace těchto rozšíření do stávající aplikace nebyla složitá.
Práce přímo nenavazuje na žádnou diplomovou práci v aktuálním ročníku, ale je možné si

představit, že by některé implementace *Ray-Castingu*, které tvoří mí kolegové, bylo možné modifikovat tak, aby zobrazovaly objemová data.
Celá práce byla tvořena za neustálého studování materiálů (jež jsou uvedeny v seznamu literatury). Zejména v období implementace bylo využito častých konzultací s vedoucím diplomové práce, které výrazným způsobem urychlily vývoj aplikace.
Realistická animace kouře je jednou z důležitých aplikací výpočetního řešení proudění v kapalinách. V praxi se těchto metod v současnosti používá například pro animaci kouře ve filmech v situacích kdy by tyto efekty byly přílíš nákladné nebo dokonce neproveditelné. Od klasického výpočetního řešení proudění v kapalinách se tyto metody odlišují hlavně tím, že není požadována vysoká míra fyzikální realističnosti, čímž se otevírá cesta pro zjednodušené modelování některých jevů (a tím také pro zrychlení těchto algoritmů). Pro zajištění maximální vizuální kvality zobrazeného kouře je nutné implementovat specializovaný volumetrický renderer, který bere v potaz specifické vlastnosti kouře.
Celkově se během práce podařilo splnit všechny body zadání a projekt považuji za úspěšně vyřešený. Projekt jako takový může být také základem pro zadání dalších prací, které by například mohly implementovat navrhovaná rozšíření. To by bylo velice vhodné, jelikož se jedná o velice zajímavou disciplínu, jejíž vývoj neustále pokračuje.

Literatura

[1] Volume ray casting. http://en.wikipedia.org/wiki/Volume_ray_casting.

[2] Voxel graphics and volume rendering i. http://prosjekt.ffi.no/unik-4660/lectures04/chapters/Voxel1.html.

[3] Y. Dobashi, K. Kaneda, T. Okita, and T. Nishita. A simple, efficient method for realistic animation of clouds. *SIGGRAPH 2000*, 2000.

[4] R. Fedkiw, J. Stam, and H. Jensen. Visual simulation of smoke. *SIGGRAPH 01*, 2001.

[5] L. Filho, M. Nussenzveig, and L. Tadmor. Approximate solution of the incompressible euler equations with no concentrations.

[6] N. Foster and D. Metaxas. Realistic animation of liquids. *Graphical Models and Image Processing (471-483)*, 1996.

[7] N. Foster and D. Metaxas. Modeling the motion of hot, turbulent gas. *SIGGRAPH 97*, 1997.

[8] J.T. Kajiya and B.P. von Herzen. Ray tracing volume densities. *SIGGRAPH 84*, 1984.

[9] Martin Schweiger. Orbiter - space flight simulator. http://orbit.medphys.ucl.ac.uk/.

[10] Andrew Selle. Fluid simulator. http://www.upl.cs.wisc.edu/ aselle/fluid/.

[11] Julian Smart, Robert Roebling, Vadim Zeitlin, and Robin Dunn. wxwidgets. `http://wxwindows.org/`.

[12] J. Stam. Real-time fluid dynamics for games, 2003.

[13] J. Steinhoff and D. Underhill. Modification of the euler equations for 'vorticity confinement'. *Physics of Fluids (2738 - 2744)*, 1994.

Dodatek A

Ovládání aplikace a ukázky výstupu

Jak již bylo uvedeno v kapitole 5 *Návrh a implementace*, tak aplikace je konzolová (její parametry se dají ovlivňovat argumenty z příkazové řádky) s možností ovlivňovat parametry za běhu programu pomocí klavesnice.
V následujícím seznamu si projdeme argumenty příkazové řádky:

-init (long) (long) (long) povinné nastavení velikosti simulačního prostoru (pro většinu příkladů byl použit krychlový prostor o straně 29 voxelů)

-slices (long) počet řezů prostorem, použitých metodou *Texture Mapping* (viz. kapitola 3 *Zobrazení objemových dat*)

-grav (double) míra gravitace (nebo také "těžkost" kouře), obvyklé nastavení je v intervalu 0.0 až 1.0

-buoy (double) míra vznosnosti (nebo také "teplota" vzduchu), obvyklé nastavení je v intervalu 0.0 až 1.0

-timestep (double) velikost časového kroku, obvyklé hodnoty jsou 0.05 až 0.5

-relsteps (int) počet kroků *Gauss—Seidelovy* relaxační metody, výchozí nastavení je 20

-vorticity (double) míra výřivosti vzduchu (více viz. sekce 5.2 nebo kapitola 4), obvyklé nastavení 0.05 až 0.5 (v závislosti na velikosti simulačního prostor)

-force (double) jednotná velikost všech externích sil, které jsou pro ovládání simulace použity

-output (string) řetězec určující jméno výstupního souboru (výchozí nastavení je "out.avi"), je velice vhodné dodržet koncovku *avi*

-h tento přepínač slouží pro tisk nápovědy do konzole

Dále budeme pokračovat výčtem kláves, které nám umožňují ovlivňovat parametry simulace za jejího běhu [1] :

'p' pozastavení a znovu-spuštění simulace

[1] V případě dvojic kláves ta první hodnotu zvyšuje a druhá snižuje

'1', 'q' změna počtu řezů použitých metodou *Texture Mapping*

'2', 'w' změna míry gravitace

'3', 'e' změna míry vznosnosti

'4', 'r' změna velikosti časového kroku

'5', 't' změna počtu relaxačních kroků *Gauss—Seidelovy* relaxační metody

'6', 'y' změna míry vířivosti

'space' jednorázové přidání externí síly, mířící vzhůru, na "dno" simulačního prostoru

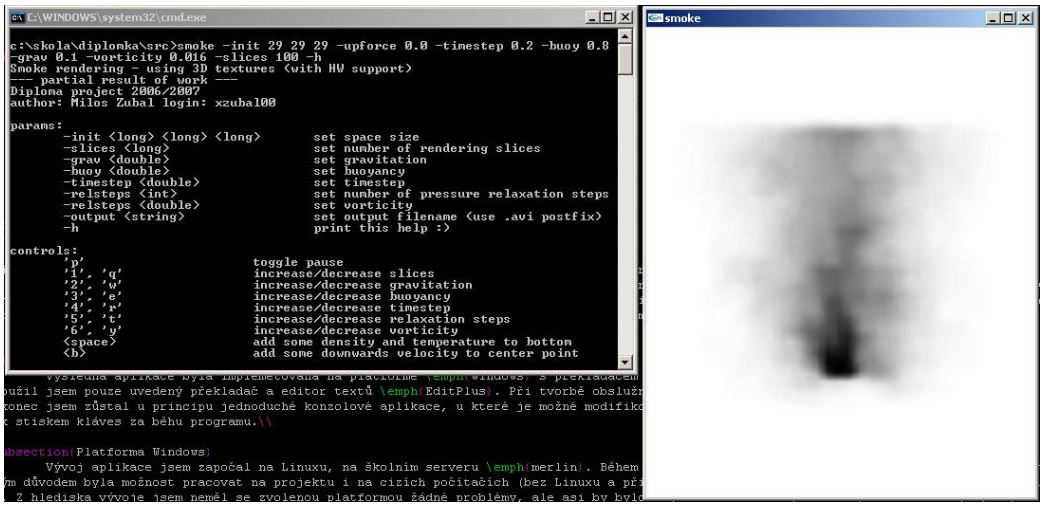
'b' jednorázové přidání externí síly, mířící dolů, do středu simulačního prostoru

'v' jednorázové přidání externí síly, mířící vlevo, do středu simulačního prostoru

'n' jednorázové přidání externí síly, mířící vpravo, do středu simulačního prostoru

'g' jednorázové přidání externí síly, mířící nahoru, do středu simulačního prostoru

To jak program vypadá za běhu je možné prohlédnout na obrázku A.1.

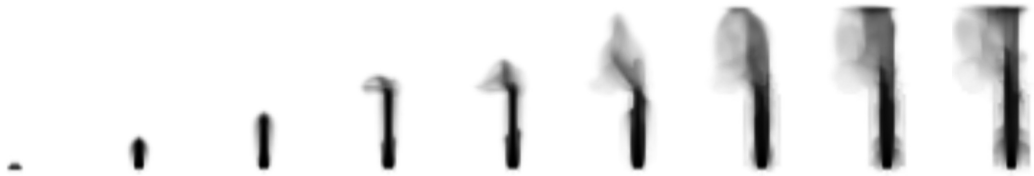


Obrázek A.1: Ukázka konzolového okna aplikace s parametry programu a nápovědou.

Jak je patrné, tak ovládání není vůbec složité. Vše bude demonstrováno na krátkém příkladě.
Nejdříve spustíme simulaci:

```
smoke -init 60 60 10 -force 10.0 -timestep 0.2 -buoy 0.7 -grav 0.06
      -vorticity 0.005 -output 2D.avi
```

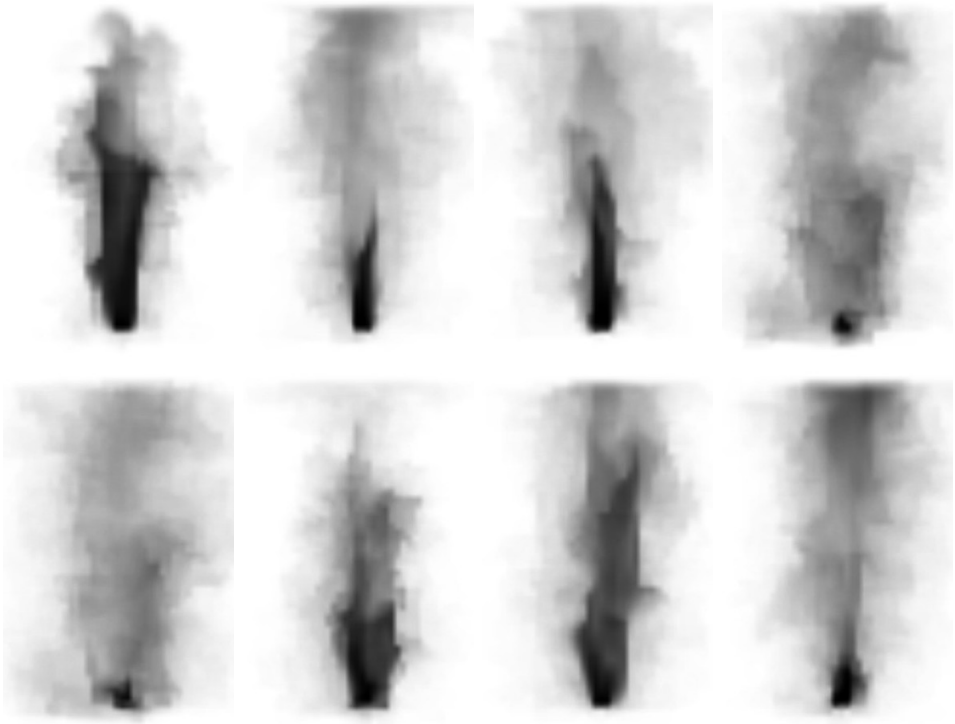
Z příkazové řádky například vidíme, že používáme poměrně nízkou míru vířivosti (což je vhodné u větších simulačních prostorů). Dále je zřetelné, že chceme modelovat lehký stoupající vzduch, protože parametr vznosnosti je cca. 10x vyšší než parametr gravitace.
V tento moment můžeme počkat až kouř dorazí do poloviny prostoru (vertikálně) a přidat



Obrázek A.2: Ukázka výstupu podle výše uvedeného postupu ovládání program.

externí sílu mířící do leva klávesou 'v'. Jak je zřetelné z obrázku A.2, tak kouř je na chvíli skutečně ovlivněn externí silou a je "tlačen" doleva.

Uživateli je doporučeno s programem experimentovat. Výsledky takových experimentů mohou vypadat podobně jako na obrázku A.3.



Obrázek A.3: Ukázka některých fází simulace při experimentování s programem.